You Dont Know JS

Riley Gelwicks

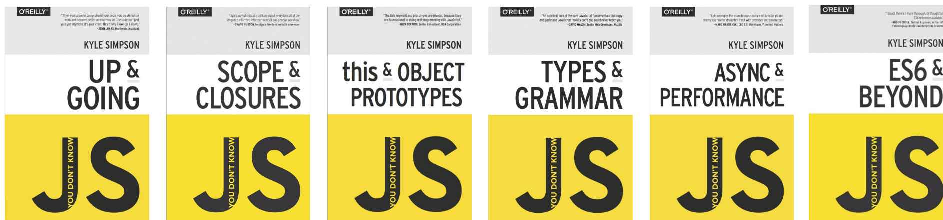
Published with GitBook



Table of Contents

You Don't Know JS (book series)

This is a series of books diving deep into the core mechanisms of the JavaScript language. The first edition of the series is now complete.



Please feel free to contribute to the quality of this content by submitting PR's for improvements to code snippets, explanations, etc. While typo fixes are welcomed, they will likely be caught through normal editing processes, and are thus not necessarily as important for this repository.

To read more about the motivations and perspective behind this book series, check out the Preface.

Titles

- Read online (free!): "Up & Going", Published: Buy Now in print, but the ebook format is free!
- Read online (free!): "Scope & Closures", Published: Buy Now
- Read online (free!): "this & Object Prototypes", Published: Buy Now
- Read online (free!): "Types & Grammar", Published: Buy Now
- Read online (free!): "Async & Performance", Published: Buy Now
- Read online (free!): "ES6 & Beyond", Published: Buy Now

Publishing

These books are being released here as drafts, free to read, but are also being edited, produced, and published through O'Reilly.

If you like the content you find here, and want to support more content like it, please purchase the books once they are available for sale, through your normal book sources. :)

If you'd like to contribute financially towards the effort (or any of my other OSS work) aside from purchasing the books, I do have a patreon that I would always appreciate your generosity towards.



In-person Training

The content for these books derives heavily from a series of training materials I teach professionally (in both public and private-corporate workshop format), called "Advanced JS: The 'What You Need To Know' Parts".

If you like this content and would like to contact me regarding conducting training on these, or other various JS/HTML5/node.js topics, please reach out to me through any of these channels listed here:

http://getify.me

Online Video Training

I also have some JS training material available in on-demand video format. I teach courses through Frontend Masters, like my Advanced JS workshop (more courses coming soon!).

That same course is also available through Pluralsight.

Content Contributions

Any contributions you make to this effort **are of course greatly appreciated**.

However, if you choose to contribute content (not just typo corrections) to this repo, you agree that you're giving me a non-exclusive license to use that content for the book series, as I (and O'Reilly) deem appropriate. You probably guessed that already, but we just have to make the lawyers happy by explicitly stating it.

So: blah, blah, blah... :)

License & Copyright

You Don't Know JS

Preface

I'm sure you noticed, but "JS" in the book series title is not an abbreviation for words used to curse about JavaScript, though cursing at the language's quirks is something we can probably all identify with!

From the earliest days of the web, JavaScript has been a foundational technology that drives interactive experience around the content we consume. While flickering mouse trails and annoying pop-up prompts may be where JavaScript started, nearly 2 decades later, the technology and capability of JavaScript has grown many orders of magnitude, and few doubt its importance at the heart of the world's most widely available software platform: the web.

But as a language, it has perpetually been a target for a great deal of criticism, owing partly to its heritage but even more to its design philosophy. Even the name evokes, as Brendan Eich once put it, "dumb kid brother" status next to its more mature older brother "Java". But the name is merely an accident of politics and marketing. The two languages are vastly different in many important ways. "JavaScript" is as related to "Java" as "Carnival" is to "Car".

Because JavaScript borrows concepts and syntax idioms from several languages, including proud C-style procedural roots as well as subtle, less obvious Scheme/Lisp-style functional roots, it is exceedingly approachable to a broad audience of developers, even those with just little to no programming experience. The "Hello World" of JavaScript is so simple that the language is inviting and easy to get comfortable with in early exposure.

While JavaScript is perhaps one of the easiest languages to get up and running with, its eccentricities make solid mastery of the language a vastly less common occurrence than in many other languages. Where it takes a pretty in-depth knowledge of a language like C or C++ to write a full-scale program, full-scale production JavaScript can, and often does, barely scratch the surface of what the language can do.

Sophisticated concepts which are deeply rooted into the language tend instead to surface themselves in *seemingly* simplistic ways, such as passing around functions as callbacks, which encourages the JavaScript developer to just use the language as-is and not worry too much about what's going on under the hood.

It is simultaneously a simple, easy-to-use language that has broad appeal, and a complex and nuanced collection of language mechanics which without careful study will elude *true understanding* even for the most seasoned of JavaScript developers.

Therein lies the paradox of JavaScript, the Achilles' Heel of the language, the challenge we are presently addressing. Because JavaScript *can* be used without understanding, the understanding of the language is often never attained.

Mission

If at every point that you encounter a surprise or frustration in JavaScript, your response is to add it to the blacklist, as some are accustomed to doing, you soon will be relegated to a hollow shell of the richness of JavaScript.

While this subset has been famously dubbed "The Good Parts", I would implore you, dear reader, to instead consider it the "The Easy Parts", "The Safe Parts", or even "The Incomplete Parts".

This *You Don't Know JavaScript* book series offers a contrary challenge: learn and deeply understand *all* of JavaScript, even and especially "The Tough Parts".

Here, we address head on the tendency of JS developers to learn "just enough" to get by, without ever forcing themselves to learn exactly how and why the language behaves the way it does. Furthermore, we eschew the common advice to *retreat* when the road gets rough.

I am not content, nor should you be, at stopping once something *just works*, and not really knowing *why*. I gently challenge you to journey down that bumpy "road less traveled" and embrace all that JavaScript is and can do. With that knowledge, no technique, no framework, no popular buzzword acronym of the week, will be beyond your understanding.

These books each take on specific core parts of the language which are most commonly misunderstood or under-understood, and dive very deep and exhaustively into them. You should come away from reading with a firm confidence in your understanding, not just of the theoretical, but the practical "what you need to know" bits.

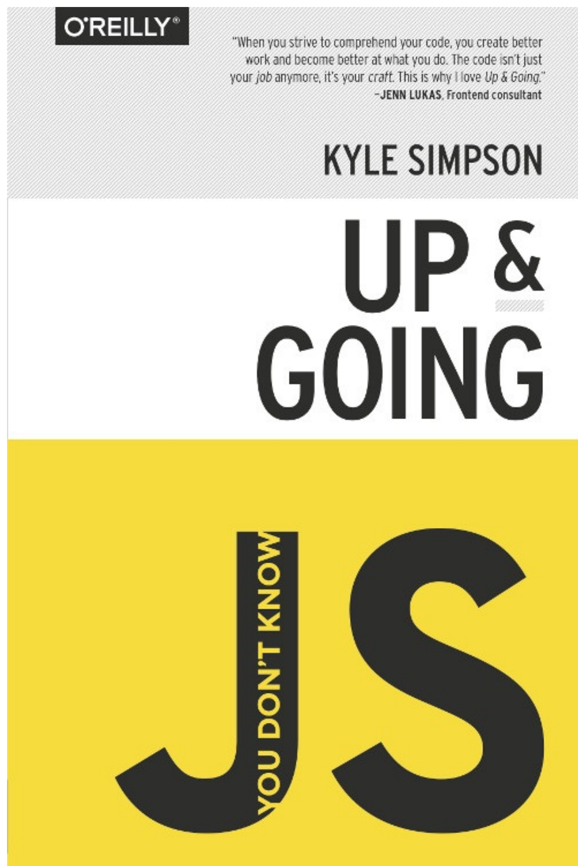
The JavaScript you know *right now* is probably *parts* handed down to you by others who've been burned by incomplete understanding. *That* JavaScript is but a shadow of the true language. You don't *really* know JavaScript, *yet*, but if you dig into this series, you *will*. Read on, my friends. JavaScript awaits you.

Summary

JavaScript is awesome. It's easy to learn partially, and much harder to learn completely (or even *sufficiently*). When developers encounter confusion, they usually blame the language instead of their lack of understanding. These books aim to fix that, inspiring a strong appreciation for the language you can now, and *should*, deeply *know*.

Note: Many of the examples in this book assume modern (and future-reaching) JavaScript engine environments, such as ES6. Some code may not work as described if run in older (pre-ES6) engines.

You Don't Know JS: Up & Going



Purchase digital/print copy from O'Reilly

Table of Contents

You Don't Know JS: Up & Going

Foreword

What was the last new thing you learned?

Perhaps it was a foreign language, like Italian or German. Or maybe it was a graphics editor, like Photoshop. Or a cooking technique or woodworking or an exercise routine. I want you to remember that feeling when you finally got it: the lightbulb moment. When things went from blurry to crystal clear, as you mastered the table saw or understood the difference between masculine and feminine nouns in French. How did it feel? Pretty amazing, right?

Now I want you to travel back a little bit further in your memory to right before you learned your new skill. How did *that* feel? Probably slightly intimidating and maybe a little bit frustrating, right? At one point, we all did not know the things that we know now and that's totally OK; we all start somewhere. Learning new material is an exciting adventure, especially if you are looking to learn the subject efficiently.

I teach a lot of beginner coding classes. The students who take my classes have often tried teaching themselves subjects like HTML or JavaScript by reading blog posts or copying and pasting code, but they haven't been able to truly master the material that will allow them to code their desired outcome. And because they don't truly grasp the ins and outs of certain coding topics, they can't write powerful code or debug their own work, as they don't really understand what is happening.

I always believe in teaching my classes the proper way, meaning I teach web standards, semantic markup, well-commented code, and other best practices. I cover the subject in a thorough manner to explain the hows and whys, without just tossing out code to copy and paste. When you strive to comprehend your code, you create better work and become better at what you do. The code isn't just your *job* anymore, it's your *craft*. This is why I love *Up & Going*. Kyle takes us on a deep dive through syntax and terminology to give a great introduction to JavaScript without cutting corners. This book doesn't skim over the surface, but really allows us to genuinely understand the concepts we will be writing.

Because it's not enough to be able to duplicate jQuery snippets into your website, the same way it's not enough to learn how to open, close, and save a document in Photoshop. Sure, once I learn a few basics about the program I could create and share a design I made. But without legitimately knowing the tools and what is behind them, how can I define a grid, or craft a legible type system, or optimize graphics for web use. The same goes for JavaScript.

Without knowing how loops work, or how to define variables, or what scope is, we won't be writing the best code we can. We don't want to settle for anything less -- this is, after all, our craft.

The more you are exposed to JavaScript, the clearer it becomes. Words like closures, objects, and methods might seem out of reach to you now, but this book will help those terms come into clarity. I want you to keep those two feelings of before and after you learn something in mind as you begin this book. It might seem daunting, but you've picked up this book because you are starting an awesome journey to hone your knowledge. *Up & Going* is the start of our path to understanding programming. Enjoy the lightbulb moments!

Jenn Lukas
jennlukas.com, @jennlukas
Front-end consultant

You Don't Know JS: Up & Going

Chapter 1: Into Programming

Welcome to the *You Don't Know JS* (*YDKJS*) series.

Up & Going is an introduction to several basic concepts of programming -- of course we lean toward JavaScript (often abbreviated JS) specifically -- and how to approach and understand the rest of the titles in this series. Especially if you're just getting into programming and/or JavaScript, this book will briefly explore what you need to get *up and going*.

This book starts off explaining the basic principles of programming at a very high level. It's mostly intended if you are starting *YDKJS* with little to no prior programming experience, and are looking to these books to help get you started along a path to understanding programming through the lens of JavaScript.

Chapter 1 should be approached as a quick overview of the things you'll want to learn more about and practice to get *into programming*. There are also many other fantastic programming introduction resources that can help you dig into these topics further, and I encourage you to learn from them in addition to this chapter.

Once you feel comfortable with general programming basics, Chapter 2 will help guide you to a familiarity with JavaScript's flavor of programming. Chapter 2 introduces what JavaScript is about, but again, it's not a comprehensive guide -- that's what the rest of the *YDKJS* books are for!

If you're already fairly comfortable with JavaScript, first check out Chapter 3 as a brief glimpse of what to expect from *YDKJS*, then jump right in!

Code

Let's start from the beginning.

A program, often referred to as *source code* or just *code*, is a set of special instructions to tell the computer what tasks to perform. Usually code is saved in a text file, although with JavaScript you can also type code directly into a developer console in a browser, which we'll cover shortly.

The rules for valid format and combinations of instructions is called a *computer language*, sometimes referred to as its *syntax*, much the same as English tells you how to spell words and how to create valid sentences using words and punctuation.

Statements

In a computer language, a group of words, numbers, and operators that performs a specific task is a *statement*. In JavaScript, a statement might look as follows:

```
a = b * 2;
```

The characters `a` and `b` are called *variables* (see "Variables"), which are like simple boxes you can store any of your stuff in. In programs, variables hold values (like the number `42`) to be used by the program. Think of them as symbolic placeholders for the values themselves.

By contrast, the `2` is just a value itself, called a *literal value*, because it stands alone without being stored in a variable.

The `=` and `*` characters are *operators* (see "Operators") -- they perform actions with the values and variables such as assignment and mathematic multiplication.

Most statements in JavaScript conclude with a semicolon (`;`) at the end.

The statement `a = b * 2;` tells the computer, roughly, to get the current value stored in the variable `b`, multiply that value by `2`, then store the result back into another variable we call `a`.

Programs are just collections of many such statements, which together describe all the steps that it takes to perform your program's purpose.

Expressions

Statements are made up of one or more *expressions*. An expression is any reference to a variable or value, or a set of variable(s) and value(s) combined with operators.

For example:

```
a = b * 2;
```

This statement has four expressions in it:

- `2` is a *literal value expression*

- `b` is a *variable expression*, which means to retrieve its current value
- `b * 2` is an *arithmetic expression*, which means to do the multiplication
- `a = b * 2` is an *assignment expression*, which means to assign the result of the `b * 2` expression to the variable `a` (more on assignments later)

A general expression that stands alone is also called an *expression statement*, such as the following:

```
b * 2;
```

This flavor of expression statement is not very common or useful, as generally it wouldn't have any effect on the running of the program -- it would retrieve the value of `b` and multiply it by `2`, but then wouldn't do anything with that result.

A more common expression statement is a *call expression* statement (see "Functions"), as the entire statement is the function call expression itself:

```
alert( a );
```

Executing a Program

How do those collections of programming statements tell the computer what to do? The program needs to be *executed*, also referred to as *running the program*.

Statements like `a = b * 2` are helpful for developers when reading and writing, but are not actually in a form the computer can directly understand. So a special utility on the computer (either an *interpreter* or a *compiler*) is used to translate the code you write into commands a computer can understand.

For some computer languages, this translation of commands is typically done from top to bottom, line by line, every time the program is run, which is usually called *interpreting* the code.

For other languages, the translation is done ahead of time, called *compiling* the code, so when the program *runs* later, what's running is actually the already compiled computer instructions ready to go.

It's typically asserted that JavaScript is *interpreted*, because your JavaScript source code is processed each time it's run. But that's not entirely accurate. The JavaScript engine actually *compiles* the program on the fly and then immediately runs the compiled code.

Note: For more information on JavaScript compiling, see the first two chapters of the *Scope & Closures* title of this series.

Try It Yourself

This chapter is going to introduce each programming concept with simple snippets of code, all written in JavaScript (obviously!).

It cannot be emphasized enough: while you go through this chapter -- and you may need to spend the time to go over it several times -- you should practice each of these concepts by typing the code yourself. The easiest way to do that is to open up the developer tools console in your nearest browser (Firefox, Chrome, IE, etc.).

Tip: Typically, you can launch the developer console with a keyboard shortcut or from a menu item. For more detailed information about launching and using the console in your favorite browser, see "Mastering The Developer Tools Console" (http://blog.teamtreehouse.com/mastering-developer-tools-console). To type multiple lines into the console at once, use `<shift> + <enter>` to move to the next new line. Once you hit `<enter>` by itself, the console will run everything you've just typed.

Let's get familiar with the process of running code in the console. First, I suggest opening up an empty tab in your browser. I prefer to do this by typing `about:blank` into the address bar. Then, make sure your developer console is open, as we just mentioned.

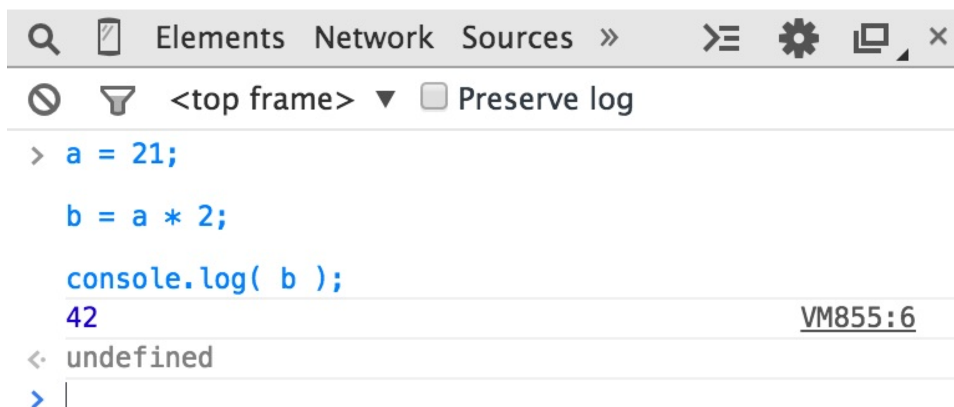
Now, type this code and see how it runs:

```
a = 21;

b = a * 2;

console.log( b );
```

Typing the preceding code into the console in Chrome should produce something like the following:



Go on, try it. The best way to learn programming is to start coding!

Output

In the previous code snippet, we used `console.log(..)`. Briefly, let's look at what that line of code is all about.

You may have guessed, but that's exactly how we print text (aka *output* to the user) in the developer console. There are two characteristics of that statement that we should explain.

First, the `log( b )` part is referred to as a function call (see "Functions"). What's happening is we're handing the `b` variable to that function, which asks it to take the value of `b` and print it to the console.

Second, the `console.` part is an object reference where the `log(..)` function is located. We cover objects and their properties in more detail in Chapter 2.

Another way of creating output that you can see is to run an `alert(..)` statement. For example:

```
alert( b );
```

If you run that, you'll notice that instead of printing the output to the console, it shows a popup "OK" box with the contents of the `b` variable. However, using `console.log(..)` is generally going to make learning about coding and running your programs in the console easier than using `alert(..)`, because you can output many values at once without interrupting the browser interface.

For this book, we'll use `console.log(..)` for output.

Input

While we're discussing output, you may also wonder about *input* (i.e., receiving information from the user).

The most common way that happens is for the HTML page to show form elements (like text boxes) to a user that they can type into, and then using JS to read those values into your program's variables.

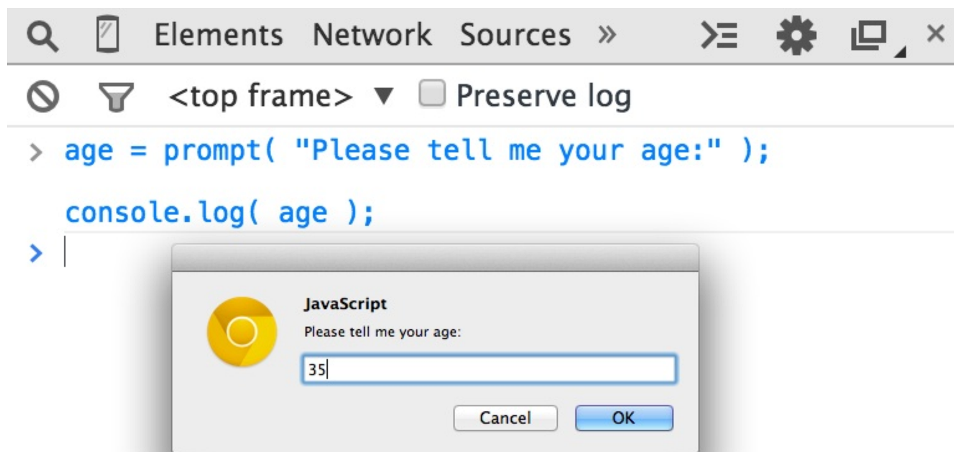
But there's an easier way to get input for simple learning and demonstration purposes such as what you'll be doing throughout this book. Use the `prompt(..)` function:

```
age = prompt( "Please tell me your age:" );

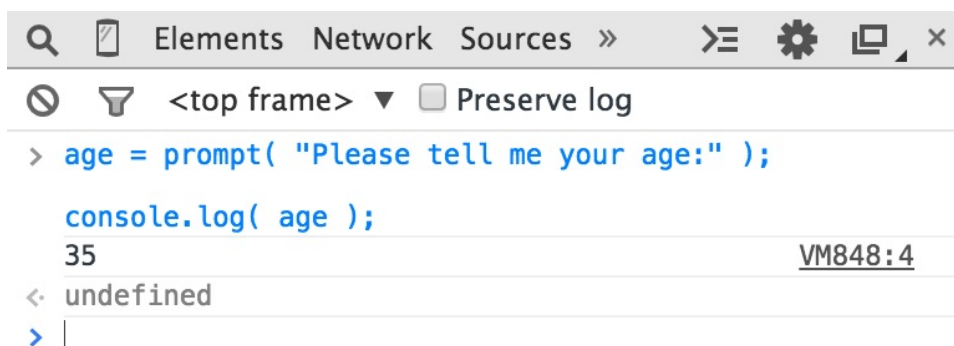
console.log( age );
```

As you may have guessed, the message you pass to `prompt(..)` -- in this case, `"Please tell me your age:"` -- is printed into the popup.

This should look similar to the following:



Once you submit the input text by clicking "OK," you'll observe that the value you typed is stored in the `age` variable, which we then *output* with `console.log(..)` :



To keep things simple while we're learning basic programming concepts, the examples in this book will not require input. But now that you've seen how to use `prompt(..)` , if you want to challenge yourself you can try to use input in your explorations of the examples.

Operators

Operators are how we perform actions on variables and values. We've already seen two JavaScript operators, the `=` and the `*` .

The `*` operator performs mathematic multiplication. Simple enough, right?

The `=` equals operator is used for *assignment* -- we first calculate the value on the *right-hand side* (source value) of the `=` and then put it into the variable that we specify on the *left-hand side* (target variable).

Warning: This may seem like a strange reverse order to specify assignment. Instead of `a = 42`, some might prefer to flip the order so the source value is on the left and the target variable is on the right, like `42 -> a` (this is not valid JavaScript!). Unfortunately, the `a = 42` ordered form, and similar variations, is quite prevalent in modern programming languages. If it feels unnatural, just spend some time rehearsing that ordering in your mind to get accustomed to it.

Consider:

```
a = 2;
b = a + 1;
```

Here, we assign the `2` value to the `a` variable. Then, we get the value of the `a` variable (still `2`), add `1` to it resulting in the value `3`, then store that value in the `b` variable.

While not technically an operator, you'll need the keyword `var` in every program, as it's the primary way you *declare* (aka *create*) *var*iables (see "Variables").

You should always declare the variable by name before you use it. But you only need to declare a variable once for each *scope* (see "Scope"); it can be used as many times after that as needed. For example:

```
var a = 20;

a = a + 1;
a = a * 2;

console.log( a );    // 42
```

Here are some of the most common operators in JavaScript:

- Assignment: `=` as in `a = 2`.
- Math: `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division), as in `a * 3`.
- Compound Assignment: `+=`, `-=`, `*=`, and `/=` are compound operators that combine a math operation with assignment, as in `a += 2` (same as `a = a + 2`).
- Increment/Decrement: `++` (increment), `--` (decrement), as in `a++` (similar to `a = a + 1`).
- Object Property Access: `.` as in `console.log()`.

  Objects are values that hold other values at specific named locations called properties. `obj.a` means an object value called `obj` with a property of the name `a`. Properties can alternatively be accessed as `obj["a"]`. See Chapter 2.

- Equality: `==` (loose-equals), `===` (strict-equals), `!=` (loose not-equals), `!==` (strict not-equals), as in `a == b`.

  See "Values & Types" and Chapter 2.

- Comparison: `<` (less than), `>` (greater than), `<=` (less than or loose-equals), `>=` (greater than or loose-equals), as in `a <= b`.

  See "Values & Types" and Chapter 2.

- Logical: `&&` (and), `||` (or), as in `a || b` that selects either `a` *or* `b`.

  These operators are used to express compound conditionals (see "Conditionals"), like if either `a` *or* `b` is true.

Note: For much more detail, and coverage of operators not mentioned here, see the Mozilla Developer Network (MDN)'s "Expressions and Operators" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators).

Values & Types

If you ask an employee at a phone store how much a certain phone costs, and they say "ninety-nine, ninety-nine" (i.e., $99.99), they're giving you an actual numeric dollar figure that represents what you'll need to pay (plus taxes) to buy it. If you want to buy two of those phones, you can easily do the mental math to double that value to get $199.98 for your base cost.

If that same employee picks up another similar phone but says it's "free" (perhaps with air quotes), they're not giving you a number, but instead another kind of representation of your expected cost ($0.00) -- the word "free."

When you later ask if the phone includes a charger, that answer could only have been either "yes" or "no."

In very similar ways, when you express values in a program, you choose different representations for those values based on what you plan to do with them.

These different representations for values are called *types* in programming terminology. JavaScript has built-in types for each of these so called *primitive* values:

- When you need to do math, you want a `number`.
- When you need to print a value on the screen, you need a `string` (one or more characters, words, sentences).
- When you need to make a decision in your program, you need a `boolean` (`true` or `false`).

Values that are included directly in the source code are called *literals*. `string` literals are surrounded by double quotes `"..."` or single quotes (`'...'`) -- the only difference is stylistic preference. `number` and `boolean` literals are just presented as is (i.e., `42` , `true` , etc.).

Consider:

```
"I am a string";
'I am also a string';

42;

true;
false;
```

Beyond `string` / `number` / `boolean` value types, it's common for programming languages to provide *arrays*, *objects*, *functions*, and more. We'll cover much more about values and types throughout this chapter and the next.

Converting Between Types

If you have a `number` but need to print it on the screen, you need to convert the value to a `string` , and in JavaScript this conversion is called "coercion." Similarly, if someone enters a series of numeric characters into a form on an ecommerce page, that's a `string` , but if you need to then use that value to do math operations, you need to *coerce* it to a `number` .

JavaScript provides several different facilities for forcibly coercing between *types*. For example:

```
var a = "42";
var b = Number( a );

console.log( a );    // "42"
console.log( b );    // 42
```

Using `Number(..)` (a built-in function) as shown is an *explicit* coercion from any other type to the `number` type. That should be pretty straightforward.

But a controversial topic is what happens when you try to compare two values that are not already of the same type, which would require *implicit* coercion.

When comparing the string `"99.99"` to the number `99.99` , most people would agree they are equivalent. But they're not exactly the same, are they? It's the same value in two different representations, two different *types*. You could say they're "loosely equal," couldn't

you?

To help you out in these common situations, JavaScript will sometimes kick in and *implicitly* coerce values to the matching types.

So if you use the `==` loose equals operator to make the comparison `"99.99" == 99.99`, JavaScript will convert the left-hand side `"99.99"` to its `number` equivalent `99.99`. The comparison then becomes `99.99 == 99.99`, which is of course `true`.

While designed to help you, implicit coercion can create confusion if you haven't taken the time to learn the rules that govern its behavior. Most JS developers never have, so the common feeling is that implicit coercion is confusing and harms programs with unexpected bugs, and should thus be avoided. It's even sometimes called a flaw in the design of the language.

However, implicit coercion is a mechanism that *can be learned*, and moreover *should be learned* by anyone wishing to take JavaScript programming seriously. Not only is it not confusing once you learn the rules, it can actually make your programs better! The effort is well worth it.

Note: For more information on coercion, see Chapter 2 of this title and Chapter 4 of the *Types & Grammar* title of this series.

Code Comments

The phone store employee might jot down some notes on the features of a newly released phone or on the new plans her company offers. These notes are only for the employee -- they're not for customers to read. Nevertheless, these notes help the employee do her job better by documenting the hows and whys of what she should tell customers.

One of the most important lessons you can learn about writing code is that it's not just for the computer. Code is every bit as much, if not more, for the developer as it is for the compiler.

Your computer only cares about machine code, a series of binary 0s and 1s, that comes from *compilation*. There's a nearly infinite number of programs you could write that yield the same series of 0s and 1s. The choices you make about how to write your program matter -- not only to you, but to your other team members and even to your future self.

You should strive not just to write programs that work correctly, but programs that make sense when examined. You can go a long way in that effort by choosing good names for your variables (see "Variables") and functions (see "Functions").

But another important part is code comments. These are bits of text in your program that are inserted purely to explain things to a human. The interpreter/compiler will always ignore these comments.

There are lots of opinions on what makes well-commented code; we can't really define absolute universal rules. But some observations and guidelines are quite useful:

- Code without comments is suboptimal.
- Too many comments (one per line, for example) is probably a sign of poorly written code.
- Comments should explain *why*, not *what*. They can optionally explain *how* if that's particularly confusing.

In JavaScript, there are two types of comments possible: a single-line comment and a multiline comment.

Consider:

```
// This is a single-line comment

/* But this is
       a multiline
             comment.
                      */
```

The `//` single-line comment is appropriate if you're going to put a comment right above a single statement, or even at the end of a line. Everything on the line after the `//` is treated as the comment (and thus ignored by the compiler), all the way to the end of the line. There's no restriction to what can appear inside a single-line comment.

Consider:

```
var a = 42;        // 42 is the meaning of life
```

The `/* .. */` multiline comment is appropriate if you have several lines worth of explanation to make in your comment.

Here's a common usage of multiline comments:

```
/* The following value is used because
   it has been shown that it answers
   every question in the universe. */
var a = 42;
```

It can also appear anywhere on a line, even in the middle of a line, because the `*/` ends it. For example:

```
var a = /* arbitrary value */ 42;

console.log( a );    // 42
```

The only thing that cannot appear inside a multiline comment is a `*/`, because that would be interpreted to end the comment.

You will definitely want to begin your learning of programming by starting off with the habit of commenting code. Throughout the rest of this chapter, you'll see I use comments to explain things, so do the same in your own practice. Trust me, everyone who reads your code will thank you!

Variables

Most useful programs need to track a value as it changes over the course of the program, undergoing different operations as called for by your program's intended tasks.

The easiest way to go about that in your program is to assign a value to a symbolic container, called a *variable* -- so called because the value in this container can *vary* over time as needed.

In some programming languages, you declare a variable (container) to hold a specific type of value, such as `number` or `string`. *Static typing*, otherwise known as *type enforcement*, is typically cited as a benefit for program correctness by preventing unintended value conversions.

Other languages emphasize types for values instead of variables. *Weak typing*, otherwise known as *dynamic typing*, allows a variable to hold any type of value at any time. It's typically cited as a benefit for program flexibility by allowing a single variable to represent a value no matter what type form that value may take at any given moment in the program's logic flow.

JavaScript uses the latter approach, *dynamic typing*, meaning variables can hold values of any *type* without any *type* enforcement.

As mentioned earlier, we declare a variable using the `var` statement -- notice there's no other *type* information in the declaration. Consider this simple program:

```
var amount = 99.99;

amount = amount * 2;

console.log( amount );        // 199.98

// convert `amount` to a string, and
// add "$" on the beginning
amount = "$" + String( amount );

console.log( amount );        // "$199.98"
```

The `amount` variable starts out holding the number `99.99`, and then holds the `number` result of `amount * 2`, which is `199.98`.

The first `console.log(..)` command has to *implicitly* coerce that `number` value to a `string` to print it out.

Then the statement `amount = "$" + String(amount)` *explicitly* coerces the `199.98` value to a `string` and adds a `"$"` character to the beginning. At this point, `amount` now holds the `string` value `"$199.98"`, so the second `console.log(..)` statement doesn't need to do any coercion to print it out.

JavaScript developers will note the flexibility of using the `amount` variable for each of the `99.99`, `199.98`, and the `"$199.98"` values. Static-typing enthusiasts would prefer a separate variable like `amountStr` to hold the final `"$199.98"` representation of the value, because it's a different type.

Either way, you'll note that `amount` holds a running value that changes over the course of the program, illustrating the primary purpose of variables: managing program *state*.

In other words, *state* is tracking the changes to values as your program runs.

Another common usage of variables is for centralizing value setting. This is more typically called *constants*, when you declare a variable with a value and intend for that value to *not change* throughout the program.

You declare these *constants*, often at the top of a program, so that it's convenient for you to have one place to go to alter a value if you need to. By convention, JavaScript variables as constants are usually capitalized, with underscores `_` between multiple words.

Here's a silly example:

```
var TAX_RATE = 0.08;    // 8% sales tax

var amount = 99.99;

amount = amount * 2;

amount = amount + (amount * TAX_RATE);

console.log( amount );              // 215.9784
console.log( amount.toFixed( 2 ) );    // "215.98"
```

Note: Similar to how `console.log(..)` is a function `log(..)` accessed as an object property on the `console` value, `toFixed(..)` here is a function that can be accessed on `number` values. JavaScript `number` s aren't automatically formatted for dollars -- the engine doesn't know what your intent is and there's no type for currency. `toFixed(..)` lets us specify how many decimal places we'd like the `number` rounded to, and it produces the `string` as necessary.

The `TAX_RATE` variable is only *constant* by convention -- there's nothing special in this program that prevents it from being changed. But if the city raises the sales tax rate to 9%, we can still easily update our program by setting the `TAX_RATE` assigned value to `0.09` in one place, instead of finding many occurrences of the value `0.08` strewn throughout the program and updating all of them.

The newest version of JavaScript at the time of this writing (commonly called "ES6") includes a new way to declare *constants*, by using `const` instead of `var` :

```
// as of ES6:
const TAX_RATE = 0.08;

var amount = 99.99;

// ..
```

Constants are useful just like variables with unchanged values, except that constants also prevent accidentally changing value somewhere else after the initial setting. If you tried to assign any different value to `TAX_RATE` after that first declaration, your program would reject the change (and in strict mode, fail with an error -- see "Strict Mode" in Chapter 2).

By the way, that kind of "protection" against mistakes is similar to the static-typing type enforcement, so you can see why static types in other languages can be attractive!

Note: For more information about how different values in variables can be used in your programs, see the *Types & Grammar* title of this series.

Blocks

The phone store employee must go through a series of steps to complete the checkout as you buy your new phone.

Similarly, in code we often need to group a series of statements together, which we often call a *block*. In JavaScript, a block is defined by wrapping one or more statements inside a curly-brace pair `{ .. }`. Consider:

```
var amount = 99.99;

// a general block
{
    amount = amount * 2;
    console.log( amount );    // 199.98
}
```

This kind of standalone `{ .. }` general block is valid, but isn't as commonly seen in JS programs. Typically, blocks are attached to some other control statement, such as an `if` statement (see "Conditionals") or a loop (see "Loops"). For example:

```
var amount = 99.99;

// is amount big enough?
if (amount > 10) {          // <-- block attached to `if`
    amount = amount * 2;
    console.log( amount );    // 199.98
}
```

We'll explain `if` statements in the next section, but as you can see, the `{ .. }` block with its two statements is attached to `if (amount > 10)`; the statements inside the block will only be processed if the conditional passes.

Note: Unlike most other statements like `console.log(amount);`, a block statement does not need a semicolon (`;`) to conclude it.

Conditionals

"Do you want to add on the extra screen protectors to your purchase, for $9.99?" The helpful phone store employee has asked you to make a decision. And you may need to first consult the current *state* of your wallet or bank account to answer that question. But obviously, this is just a simple "yes or no" question.

There are quite a few ways we can express *conditionals* (aka decisions) in our programs.

The most common one is the `if` statement. Essentially, you're saying, "*If* this condition is true, do the following...". For example:

```
var bank_balance = 302.13;
var amount = 99.99;

if (amount < bank_balance) {
    console.log( "I want to buy this phone!" );
}
```

The `if` statement requires an expression in between the parentheses `( )` that can be treated as either `true` or `false`. In this program, we provided the expression `amount < bank_balance`, which indeed will either evaluate to `true` or `false` depending on the amount in the `bank_balance` variable.

You can even provide an alternative if the condition isn't true, called an `else` clause. Consider:

```
const ACCESSORY_PRICE = 9.99;

var bank_balance = 302.13;
var amount = 99.99;

amount = amount * 2;

// can we afford the extra purchase?
if ( amount < bank_balance ) {
    console.log( "I'll take the accessory!" );
    amount = amount + ACCESSORY_PRICE;
}
// otherwise:
else {
    console.log( "No, thanks." );
}
```

Here, if `amount < bank_balance` is `true`, we'll print out `"I'll take the accessory!"` and add the `9.99` to our `amount` variable. Otherwise, the `else` clause says we'll just politely respond with `"No, thanks."` and leave `amount` unchanged.

As we discussed in "Values & Types" earlier, values that aren't already of an expected type are often coerced to that type. The `if` statement expects a `boolean`, but if you pass it something that's not already `boolean`, coercion will occur.

JavaScript defines a list of specific values that are considered "falsy" because when coerced to a `boolean`, they become `false` -- these include values like `0` and `""`. Any other value not on the "falsy" list is automatically "truthy" -- when coerced to a `boolean` they become `true`. Truthy values include things like `99.99` and `"free"`. See "Truthy & Falsy" in Chapter 2 for more information.

Conditionals exist in other forms besides the `if`. For example, the `switch` statement can be used as a shorthand for a series of `if..else` statements (see Chapter 2). Loops (see "Loops") use a *conditional* to determine if the loop should keep going or stop.

Note: For deeper information about the coercions that can occur implicitly in the test expressions of *conditionals*, see Chapter 4 of the *Types & Grammar* title of this series.

Loops

During busy times, there's a waiting list for customers who need to speak to the phone store employee. While there's still people on that list, she just needs to keep serving the next customer.

Repeating a set of actions until a certain condition fails -- in other words, repeating only while the condition holds -- is the job of programming loops; loops can take different forms, but they all satisfy this basic behavior.

A loop includes the test condition as well as a block (typically as `{ .. }`). Each time the loop block executes, that's called an *iteration*.

For example, the `while` loop and the `do..while` loop forms illustrate the concept of repeating a block of statements until a condition no longer evaluates to `true`:

```
while (numOfCustomers > 0) {
    console.log( "How may I help you?" );

    // help the customer...

    numOfCustomers = numOfCustomers - 1;
}

// versus:

do {
    console.log( "How may I help you?" );

    // help the customer...

    numOfCustomers = numOfCustomers - 1;
} while (numOfCustomers > 0);
```

The only practical difference between these loops is whether the conditional is tested before the first iteration (`while`) or after the first iteration (`do..while`).

In either form, if the conditional tests as `false` , the next iteration will not run. That means if the condition is initially `false` , a `while` loop will never run, but a `do..while` loop will run just the first time.

Sometimes you are looping for the intended purpose of counting a certain set of numbers, like from `0` to `9` (ten numbers). You can do that by setting a loop iteration variable like `i` at value `0` and incrementing it by `1` each iteration.

Warning: For a variety of historical reasons, programming languages almost always count things in a zero-based fashion, meaning starting with `0` instead of `1` . If you're not familiar with that mode of thinking, it can be quite confusing at first. Take some time to practice counting starting with `0` to become more comfortable with it!

The conditional is tested on each iteration, much as if there is an implied `if` statement inside the loop.

We can use JavaScript's `break` statement to stop a loop. Also, we can observe that it's awfully easy to create a loop that would otherwise run forever without a `break` ing mechanism.

Let's illustrate:

```
var i = 0;

// a `while..true` loop would run forever, right?
while (true) {
    // stop the loop?
    if ((i <= 9) === false) {
        break;
    }

    console.log( i );
    i = i + 1;
}
// 0 1 2 3 4 5 6 7 8 9
```

Warning: This is not necessarily a practical form you'd want to use for your loops. It's presented here for illustration purposes only.

While a `while` (or `do..while`) can accomplish the task manually, there's another syntactic form called a `for` loop for just that purpose:

```
for (var i = 0; i <= 9; i = i + 1) {
    console.log( i );
}
// 0 1 2 3 4 5 6 7 8 9
```

As you can see, in both cases the conditional `i <= 9` is `true` for the first 10 iterations (`i` of values `0` through `9`) of either loop form, but becomes `false` once `i` is value `10` .

The `for` loop has three clauses: the initialization clause (`var i=0`), the conditional test clause (`i <= 9`), and the update clause (`i = i + 1`). So if you're going to do counting with your loop iterations, `for` is a more compact and often easier form to understand and write.

There are other specialized loop forms that are intended to iterate over specific values, such as the properties of an object (see Chapter 2) where the implied conditional test is just whether all the properties have been processed. The "loop until a condition fails" concept holds no matter what the form of the loop.

Functions

The phone store employee probably doesn't carry around a calculator to figure out the taxes and final purchase amount. That's a task she needs to define once and reuse over and over again. Odds are, the company has a checkout register (computer, tablet, etc.) with those "functions" built in.

Similarly, your program will almost certainly want to break up the code's tasks into reusable pieces, instead of repeatedly repeating yourself repetitiously (pun intended!). The way to do this is to define a `function`.

A function is generally a named section of code that can be "called" by name, and the code inside it will be run each time. Consider:

```
function printAmount() {
    console.log( amount.toFixed( 2 ) );
}

var amount = 99.99;

printAmount(); // "99.99"

amount = amount * 2;

printAmount(); // "199.98"
```

Functions can optionally take arguments (aka parameters) -- values you pass in. And they can also optionally return a value back.

```
function printAmount(amt) {
    console.log( amt.toFixed( 2 ) );
}

function formatAmount() {
    return "$" + amount.toFixed( 2 );
}

var amount = 99.99;

printAmount( amount * 2 );        // "199.98"

amount = formatAmount();
console.log( amount );            // "$99.99"
```

The function `printAmount(..)` takes a parameter that we call `amt`. The function `formatAmount()` returns a value. Of course, you can also combine those two techniques in the same function.

Functions are often used for code that you plan to call multiple times, but they can also be useful just to organize related bits of code into named collections, even if you only plan to call them once.

Consider:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // calculate the new amount with the tax
    amt = amt + (amt * TAX_RATE);

    // return the new amount
    return amt;
}

var amount = 99.99;

amount = calculateFinalPurchaseAmount( amount );

console.log( amount.toFixed( 2 ) );        // "107.99"
```

Although `calculateFinalPurchaseAmount(..)` is only called once, organizing its behavior into a separate named function makes the code that uses its logic (the `amount = calculateFinal...` statement) cleaner. If the function had more statements in it, the benefits would be even more pronounced.

Scope

If you ask the phone store employee for a phone model that her store doesn't carry, she will not be able to sell you the phone you want. She only has access to the phones in her store's inventory. You'll have to try another store to see if you can find the phone you're looking for.

Programming has a term for this concept: *scope* (technically called *lexical scope*). In JavaScript, each function gets its own scope. Scope is basically a collection of variables as well as the rules for how those variables are accessed by name. Only code inside that function can access that function's *scoped* variables.

A variable name has to be unique within the same scope -- there can't be two different `a` variables sitting right next to each other. But the same variable name `a` could appear in different scopes.

```
function one() {
    // this `a` only belongs to the `one()` function
    var a = 1;
    console.log( a );
}

function two() {
    // this `a` only belongs to the `two()` function
    var a = 2;
    console.log( a );
}

one();        // 1
two();        // 2
```

Also, a scope can be nested inside another scope, just like if a clown at a birthday party blows up one balloon inside another balloon. If one scope is nested inside another, code inside the innermost scope can access variables from either scope.

Consider:

```
function outer() {
    var a = 1;

    function inner() {
        var b = 2;

        // we can access both `a` and `b` here
        console.log( a + b );    // 3
    }

    inner();

    // we can only access `a` here
    console.log( a );            // 1
}

outer();
```

Lexical scope rules say that code in one scope can access variables of either that scope or any scope outside of it.

So, code inside the `inner()` function has access to both variables `a` and `b`, but code only in `outer()` has access only to `a` -- it cannot access `b` because that variable is only inside `inner()`.

Recall this code snippet from earlier:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // calculate the new amount with the tax
    amt = amt + (amt * TAX_RATE);

    // return the new amount
    return amt;
}
```

The `TAX_RATE` constant (variable) is accessible from inside the `calculateFinalPurchaseAmount(..)` function, even though we didn't pass it in, because of lexical scope.

Note: For more information about lexical scope, see the first three chapters of the *Scope & Closures* title of this series.

Practice

There is absolutely no substitute for practice in learning programming. No amount of articulate writing on my part is alone going to make you a programmer.

With that in mind, let's try practicing some of the concepts we learned here in this chapter. I'll give the "requirements," and you try it first. Then consult the code listing below to see how I approached it.

- Write a program to calculate the total price of your phone purchase. You will keep purchasing phones (hint: loop!) until you run out of money in your bank account. You'll also buy accessories for each phone as long as your purchase amount is below your mental spending threshold.
- After you've calculated your purchase amount, add in the tax, then print out the calculated purchase amount, properly formatted.
- Finally, check the amount against your bank account balance to see if you can afford it or not.
- You should set up some constants for the "tax rate," "phone price," "accessory price," and "spending threshold," as well as a variable for your "bank account balance.""
- You should define functions for calculating the tax and for formatting the price with a "$" and rounding to two decimal places.
- **Bonus Challenge:** Try to incorporate input into this program, perhaps with the `prompt(..)` covered in "Input" earlier. You may prompt the user for their bank account balance, for example. Have fun and be creative!

OK, go ahead. Try it. Don't peek at my code listing until you've given it a shot yourself!

Note: Because this is a JavaScript book, I'm obviously going to solve the practice exercise in JavaScript. But you can do it in another language for now if you feel more comfortable.

Here's my JavaScript solution for this exercise:

```
const SPENDING_THRESHOLD = 200;
const TAX_RATE = 0.08;
const PHONE_PRICE = 99.99;
const ACCESSORY_PRICE = 9.99;

var bank_balance = 303.91;
var amount = 0;

function calculateTax(amount) {
    return amount * TAX_RATE;
}

function formatAmount(amount) {
    return "$" + amount.toFixed( 2 );
}

// keep buying phones while you still have money
while (amount < bank_balance) {
    // buy a new phone!
    amount = amount + PHONE_PRICE;

    // can we afford the accessory?
    if (amount < SPENDING_THRESHOLD) {
        amount = amount + ACCESSORY_PRICE;
    }
}

// don't forget to pay the government, too
amount = amount + calculateTax( amount );

console.log(
    "Your purchase: " + formatAmount( amount )
);
// Your purchase: $334.76

// can you actually afford this purchase?
if (amount > bank_balance) {
    console.log(
        "You can't afford this purchase. :("
    );
}
// You can't afford this purchase. :(
```

Note: The simplest way to run this JavaScript program is to type it into the developer console of your nearest browser.

How did you do? It wouldn't hurt to try it again now that you've seen my code. And play around with changing some of the constants to see how the program runs with different values.

Review

Learning programming doesn't have to be a complex and overwhelming process. There are just a few basic concepts you need to wrap your head around.

These act like building blocks. To build a tall tower, you start first by putting block on top of block on top of block. The same goes with programming. Here are some of the essential programming building blocks:

- You need *operators* to perform actions on values.
- You need values and *types* to perform different kinds of actions like math on `number`s or output with `string`s.
- You need *variables* to store data (aka *state*) during your program's execution.
- You need *conditionals* like `if` statements to make decisions.
- You need *loops* to repeat tasks until a condition stops being true.
- You need *functions* to organize your code into logical and reusable chunks.

Code comments are one effective way to write more readable code, which makes your program easier to understand, maintain, and fix later if there are problems.

Finally, don't neglect the power of practice. The best way to learn how to write code is to write code.

I'm excited you're well on your way to learning how to code, now! Keep it up. Don't forget to check out other beginner programming resources (books, blogs, online training, etc.). This chapter and this book are a great start, but they're just a brief introduction.

The next chapter will review many of the concepts from this chapter, but from a more JavaScript-specific perspective, which will highlight most of the major topics that are addressed in deeper detail throughout the rest of the series.

You Don't Know JS: Up & Going

Chapter 2: Into JavaScript

In the previous chapter, I introduced the basic building blocks of programming, such as variables, loops, conditionals, and functions. Of course, all the code shown has been in JavaScript. But in this chapter, we want to focus specifically on things you need to know about JavaScript to get up and going as a JS developer.

We will introduce quite a few concepts in this chapter that will not be fully explored until subsequent *YDKJS* books. You can think of this chapter as an overview of the topics covered in detail throughout the rest of this series.

Especially if you're new to JavaScript, you should expect to spend quite a bit of time reviewing the concepts and code examples here multiple times. Any good foundation is laid brick by brick, so don't expect that you'll immediately understand it all the first pass through.

Your journey to deeply learn JavaScript starts here.

Note: As I said in Chapter 1, you should definitely try all this code yourself as you read and work through this chapter. Be aware that some of the code here assumes capabilities introduced in the newest version of JavaScript at the time of this writing (commonly referred to as "ES6" for the 6th edition of ECMAScript -- the official name of the JS specification). If you happen to be using an older, pre-ES6 browser, the code may not work. A recent update of a modern browser (like Chrome, Firefox, or IE) should be used.

Values & Types

As we asserted in Chapter 1, JavaScript has typed values, not typed variables. The following built-in types are available:

- `string`
- `number`
- `boolean`
- `null` and `undefined`
- `object`
- `symbol` (new to ES6)

JavaScript provides a `typeof` operator that can examine a value and tell you what type it is:

```
var a;
typeof a;               // "undefined"

a = "hello world";
typeof a;               // "string"

a = 42;
typeof a;               // "number"

a = true;
typeof a;               // "boolean"

a = null;
typeof a;               // "object" -- weird, bug

a = undefined;
typeof a;               // "undefined"

a = { b: "c" };
typeof a;               // "object"
```

The return value from the `typeof` operator is always one of six (seven as of ES6!) string values. That is, `typeof "abc"` returns `"string"`, not `string`.

Notice how in this snippet the `a` variable holds every different type of value, and that despite appearances, `typeof a` is not asking for the "type of `a`", but rather for the "type of the value currently in `a`." Only values have types in JavaScript; variables are just simple containers for those values.

`typeof null` is an interesting case, because it errantly returns `"object"`, when you'd expect it to return `"null"`.

Warning: This is a long-standing bug in JS, but one that is likely never going to be fixed. Too much code on the Web relies on the bug and thus fixing it would cause a lot more bugs!

Also, note `a = undefined`. We're explicitly setting `a` to the `undefined` value, but that is behaviorally no different from a variable that has no value set yet, like with the `var a;` line at the top of the snippet. A variable can get to this "undefined" value state in several different ways, including functions that return no values and usage of the `void` operator.

Objects

The `object` type refers to a compound value where you can set properties (named locations) that each hold their own values of any type. This is perhaps one of the most useful value types in all of JavaScript.

```
var obj = {
    a: "hello world",
    b: 42,
    c: true
};

obj.a;          // "hello world"
obj.b;          // 42
obj.c;          // true

obj["a"];     // "hello world"
obj["b"];     // 42
obj["c"];     // true
```

It may be helpful to think of this `obj` value visually:

obj

a: "hello world" | b: 42 | c: true

Properties can either be accessed with *dot notation* (i.e., `obj.a`) or *bracket notation* (i.e., `obj["a"]`). Dot notation is shorter and generally easier to read, and is thus preferred when possible.

Bracket notation is useful if you have a property name that has special characters in it, like `obj["hello world!"]` -- such properties are often referred to as *keys* when accessed via bracket notation. The `[ ]` notation requires either a variable (explained next) or a `string` *literal* (which needs to be wrapped in `" .. "` or `' .. '`).

Of course, bracket notation is also useful if you want to access a property/key but the name is stored in another variable, such as:

```
var obj = {
    a: "hello world",
    b: 42
};

var b = "a";

obj[b];             // "hello world"
obj["b"];         // 42
```

Note: For more information on JavaScript `object` s, see the *this & Object Prototypes* title of this series, specifically Chapter 3.

There are a couple of other value types that you will commonly interact with in JavaScript programs: *array* and *function*. But rather than being proper built-in types, these should be thought of more like subtypes -- specialized versions of the `object` type.

Arrays

An array is an `object` that holds values (of any type) not particularly in named properties/keys, but rather in numerically indexed positions. For example:

```
var arr = [
    "hello world",
    42,
    true
];

arr[0];             // "hello world"
arr[1];             // 42
arr[2];             // true
arr.length;         // 3

typeof arr;         // "object"
```

Note: Languages that start counting at zero, like JS does, use `0` as the index of the first element in the array.

It may be helpful to think of `arr` visually:

arr

0: "hello world"	1: 42	2: true

Because arrays are special objects (as `typeof` implies), they can also have properties, including the automatically updated `length` property.

You theoretically could use an array as a normal object with your own named properties, or you could use an `object` but only give it numeric properties (`0` , `1` , etc.) similar to an array. However, this would generally be considered improper usage of the respective types.

The best and most natural approach is to use arrays for numerically positioned values and use `object` s for named properties.

Functions

The other `object` subtype you'll use all over your JS programs is a function:

```
function foo() {
    return 42;
}

foo.bar = "hello world";

typeof foo;             // "function"
typeof foo();         // "number"
typeof foo.bar;         // "string"
```

Again, functions are a subtype of `objects` -- `typeof` returns `"function"`, which implies that a `function` is a main type -- and can thus have properties, but you typically will only use function object properties (like `foo.bar`) in limited cases.

Note: For more information on JS values and their types, see the first two chapters of the *Types & Grammar* title of this series.

Built-In Type Methods

The built-in types and subtypes we've just discussed have behaviors exposed as properties and methods that are quite powerful and useful.

For example:

```
var a = "hello world";
var b = 3.14159;

a.length;                 // 11
a.toUpperCase();        // "HELLO WORLD"
b.toFixed(4);             // "3.1416"
```

The "how" behind being able to call `a.toUpperCase()` is more complicated than just that method existing on the value.

Briefly, there is a `String` (capital `S`) object wrapper form, typically called a "native," that pairs with the primitive `string` type; it's this object wrapper that defines the `toUpperCase()` method on its prototype.

When you use a primitive value like `"hello world"` as an `object` by referencing a property or method (e.g., `a.toUpperCase()` in the previous snippet), JS automatically "boxes" the value to its object wrapper counterpart (hidden under the covers).

A `string` value can be wrapped by a `String` object, a `number` can be wrapped by a `Number` object, and a `boolean` can be wrapped by a `Boolean` object. For the most part, you don't need to worry about or directly use these object wrapper forms of the values --

prefer the primitive value forms in practically all cases and JavaScript will take care of the rest for you.

Note: For more information on JS natives and "boxing," see Chapter 3 of the *Types & Grammar* title of this series. To better understand the prototype of an object, see Chapter 5 of the *this & Object Prototypes* title of this series.

Comparing Values

There are two main types of value comparison that you will need to make in your JS programs: *equality* and *inequality*. The result of any comparison is a strictly `boolean` value (`true` or `false`), regardless of what value types are compared.

Coercion

We talked briefly about coercion in Chapter 1, but let's revisit it here.

Coercion comes in two forms in JavaScript: *explicit* and *implicit*. Explicit coercion is simply that you can see obviously from the code that a conversion from one type to another will occur, whereas implicit coercion is when the type conversion can happen as more of a non-obvious side effect of some other operation.

You've probably heard sentiments like "coercion is evil" drawn from the fact that there are clearly places where coercion can produce some surprising results. Perhaps nothing evokes frustration from developers more than when the language surprises them.

Coercion is not evil, nor does it have to be surprising. In fact, the majority of cases you can construct with type coercion are quite sensible and understandable, and can even be used to *improve* the readability of your code. But we won't go much further into that debate -- Chapter 4 of the *Types & Grammar* title of this series covers all sides.

Here's an example of *explicit* coercion:

```
var a = "42";

var b = Number( a );

a;                // "42"
b;                // 42 -- the number!
```

And here's an example of *implicit* coercion:

```js
var a = "42";

var b = a * 1;    // "42" implicitly coerced to 42 here

a;                // "42"
b;                // 42 -- the number!
```

Truthy & Falsy

In Chapter 1, we briefly mentioned the "truthy" and "falsy" nature of values: when a non- `boolean` value is coerced to a `boolean` , does it become `true` or `false` , respectively?

The specific list of "falsy" values in JavaScript is as follows:

- `""` (empty string)
- `0` , `-0` , `NaN` (invalid `number`)
- `null` , `undefined`
- `false`

Any value that's not on this "falsy" list is "truthy." Here are some examples of those:

- `"hello"`
- `42`
- `true`
- `[ ]` , `[ 1, "2", 3 ]` (arrays)
- `{ }` , `{ a: 42 }` (objects)
- `function foo() { .. }` (functions)

It's important to remember that a non- `boolean` value only follows this "truthy"/"falsy" coercion if it's actually coerced to a `boolean` . It's not all that difficult to confuse yourself with a situation that seems like it's coercing a value to a `boolean` when it's not.

Equality

There are four equality operators: `==` , `===` , `!=` , and `!==` . The `!` forms are of course the symmetric "not equal" versions of their counterparts; *non-equality* should not be confused with *inequality*.

The difference between `==` and `===` is usually characterized that `==` checks for value equality and `===` checks for both value and type equality. However, this is inaccurate. The proper way to characterize them is that `==` checks for value equality with coercion allowed, and `===` checks for value equality without allowing coercion; `===` is often called "strict equality" for this reason.

Consider the implicit coercion that's allowed by the `==` loose-equality comparison and not allowed with the `===` strict-equality:

```
var a = "42";
var b = 42;

a == b;             // true
a === b;        // false
```

In the `a == b` comparison, JS notices that the types do not match, so it goes through an ordered series of steps to coerce one or both values to a different type until the types match, where then a simple value equality can be checked.

If you think about it, there's two possible ways `a == b` could give `true` via coercion. Either the comparison could end up as `42 == 42` or it could be `"42" == "42"`. So which is it?

The answer: `"42"` becomes `42`, to make the comparison `42 == 42`. In such a simple example, it doesn't really seem to matter which way that process goes, as the end result is the same. There are more complex cases where it matters not just what the end result of the comparison is, but *how* you get there.

The `a === b` produces `false`, because the coercion is not allowed, so the simple value comparison obviously fails. Many developers feel that `===` is more predictable, so they advocate always using that form and staying away from `==`. I think this view is very shortsighted. I believe `==` is a powerful tool that helps your program, *if you take the time to learn how it works.*

We're not going to cover all the nitty-gritty details of how the coercion in `==` comparisons works here. Much of it is pretty sensible, but there are some important corner cases to be careful of. You can read section 11.9.3 of the ES5 specification (http://www.ecma-international.org/ecma-262/5.1/) to see the exact rules, and you'll be surprised at just how straightforward this mechanism is, compared to all the negative hype surrounding it.

To boil down a whole lot of details to a few simple takeaways, and help you know whether to use `==` or `===` in various situations, here are my simple rules:

- If either value (aka side) in a comparison could be the `true` or `false` value, avoid `==` and use `===`.
- If either value in a comparison could be of these specific values (`0`, `""`, or `[]` -- empty array), avoid `==` and use `===`.
- In *all* other cases, you're safe to use `==`. Not only is it safe, but in many cases it simplifies your code in a way that improves readability.

What these rules boil down to is requiring you to think critically about your code and about what kinds of values can come through variables that get compared for equality. If you can be certain about the values, and `==` is safe, use it! If you can't be certain about the values, use `===` . It's that simple.

The `!=` non-equality form pairs with `==` , and the `!==` form pairs with `===` . All the rules and observations we just discussed hold symmetrically for these non-equality comparisons.

You should take special note of the `==` and `===` comparison rules if you're comparing two non-primitive values, like `object` s (including `function` and `array`). Because those values are actually held by reference, both `==` and `===` comparisons will simply check whether the references match, not anything about the underlying values.

For example, `array` s are by default coerced to `string` s by simply joining all the values with commas (`,`) in between. You might think that two `array` s with the same contents would be `==` equal, but they're not:

```
var a = [1,2,3];
var b = [1,2,3];
var c = "1,2,3";

a == c;         // true
b == c;         // true
a == b;         // false
```

Note: For more information about the `==` equality comparison rules, see the ES5 specification (section 11.9.3) and also consult Chapter 4 of the *Types & Grammar* title of this series; see Chapter 2 for more information about values versus references.

Inequality

The `<` , `>` , `<=` , and `>=` operators are used for inequality, referred to in the specification as "relational comparison." Typically they will be used with ordinally comparable values like `number` s. It's easy to understand that `3 < 4` .

But JavaScript `string` values can also be compared for inequality, using typical alphabetic rules (`"bar" < "foo"`).

What about coercion? Similar rules as `==` comparison (though not exactly identical!) apply to the inequality operators. Notably, there are no "strict inequality" operators that would disallow coercion the same way `===` "strict equality" does.

Consider:

```js
var a = 41;
var b = "42";
var c = "43";

a < b;		// true
b < c;		// true
```

What happens here? In section 11.8.5 of the ES5 specification, it says that if both values in the `<` comparison are `string`s, as it is with `b < c`, the comparison is made lexicographically (aka alphabetically like a dictionary). But if one or both is not a `string`, as it is with `a < b`, then both values are coerced to be `number`s, and a typical numeric comparison occurs.

The biggest gotcha you may run into here with comparisons between potentially different value types -- remember, there are no "strict inequality" forms to use -- is when one of the values cannot be made into a valid number, such as:

```js
var a = 42;
var b = "foo";

a < b;		// false
a > b;		// false
a == b;		// false
```

Wait, how can all three of those comparisons be `false`? Because the `b` value is being coerced to the "invalid number value" `NaN` in the `<` and `>` comparisons, and the specification says that `NaN` is neither greater-than nor less-than any other value.

The `==` comparison fails for a different reason. `a == b` could fail if it's interpreted either as `42 == NaN` or `"42" == "foo"` -- as we explained earlier, the former is the case.

Note: For more information about the inequality comparison rules, see section 11.8.5 of the ES5 specification and also consult Chapter 4 of the *Types & Grammar* title of this series.

Variables

In JavaScript, variable names (including function names) must be valid *identifiers*. The strict and complete rules for valid characters in identifiers are a little complex when you consider nontraditional characters such as Unicode. If you only consider typical ASCII alphanumeric characters, though, the rules are simple.

An identifier must start with `a`-`z`, `A`-`Z`, `$`, or `_`. It can then contain any of those characters plus the numerals `0`-`9`.

Generally, the same rules apply to a property name as to a variable identifier. However, certain words cannot be used as variables, but are OK as property names. These words are called "reserved words," and include the JS keywords (`for` , `in` , `if` , etc.) as well as `null` , `true` , and `false` .

Note: For more information about reserved words, see Appendix A of the *Types & Grammar* title of this series.

Function Scopes

You use the `var` keyword to declare a variable that will belong to the current function scope, or the global scope if at the top level outside of any function.

Hoisting

Wherever a `var` appears inside a scope, that declaration is taken to belong to the entire scope and accessible everywhere throughout.

Metaphorically, this behavior is called *hoisting*, when a `var` declaration is conceptually "moved" to the top of its enclosing scope. Technically, this process is more accurately explained by how code is compiled, but we can skip over those details for now.

Consider:

```
var a = 2;

foo();                    // works because `foo()`
                        // declaration is "hoisted"

function foo() {
    a = 3;

    console.log( a );    // 3

    var a;                // declaration is "hoisted"
                        // to the top of `foo()`
}

console.log( a );    // 2
```

Warning: It's not common or a good idea to rely on variable *hoisting* to use a variable earlier in its scope than its `var` declaration appears; it can be quite confusing. It's much more common and accepted to use *hoisted* function declarations, as we do with the `foo()` call appearing before its formal declaration.

Nested Scopes

When you declare a variable, it is available anywhere in that scope, as well as any lower/inner scopes. For example:

```
function foo() {
    var a = 1;

    function bar() {
        var b = 2;

        function baz() {
            var c = 3;

            console.log( a, b, c );    // 1 2 3
        }

        baz();
        console.log( a, b );        // 1 2
    }

    bar();
    console.log( a );                // 1
}

foo();
```

Notice that `c` is not available inside of `bar()`, because it's declared only inside the inner `baz()` scope, and that `b` is not available to `foo()` for the same reason.

If you try to access a variable's value in a scope where it's not available, you'll get a `ReferenceError` thrown. If you try to set a variable that hasn't been declared, you'll either end up creating a variable in the top-level global scope (bad!) or getting an error, depending on "strict mode" (see "Strict Mode"). Let's take a look:

```
function foo() {
    a = 1;    // `a` not formally declared
}

foo();
a;            // 1 -- oops, auto global variable :(
```

This is a very bad practice. Don't do it! Always formally declare your variables.

In addition to creating declarations for variables at the function level, ES6 *lets* you declare variables to belong to individual blocks (pairs of `{ .. }`), using the `let` keyword. Besides some nuanced details, the scoping rules will behave roughly the same as we just saw with

functions:

```
function foo() {
    var a = 1;

    if (a >= 1) {
        let b = 2;

        while (b < 5) {
            let c = b * 2;
            b++;

            console.log( a + c );
        }
    }
}

foo();
// 5 7 9
```

Because of using `let` instead of `var` , `b` will belong only to the `if` statement and thus not to the whole `foo()` function's scope. Similarly, `c` belongs only to the `while` loop. Block scoping is very useful for managing your variable scopes in a more fine-grained fashion, which can make your code much easier to maintain over time.

Note: For more information about scope, see the *Scope & Closures* title of this series. See the *ES6 & Beyond* title of this series for more information about `let` block scoping.

Conditionals

In addition to the `if` statement we introduced briefly in Chapter 1, JavaScript provides a few other conditionals mechanisms that we should take a look at.

Sometimes you may find yourself writing a series of `if..else..if` statements like this:

```
if (a == 2) {
    // do something
}
else if (a == 10) {
    // do another thing
}
else if (a == 42) {
    // do yet another thing
}
else {
    // fallback to here
}
```

This structure works, but it's a little verbose because you need to specify the `a` test for each case. Here's another option, the `switch` statement:

```
switch (a) {
    case 2:
        // do something
        break;
    case 10:
        // do another thing
        break;
    case 42:
        // do yet another thing
        break;
    default:
        // fallback to here
}
```

The `break` is important if you want only the statement(s) in one `case` to run. If you omit `break` from a `case`, and that `case` matches or runs, execution will continue with the next `case`'s statements regardless of that `case` matching. This so called "fall through" is sometimes useful/desired:

```
switch (a) {
    case 2:
    case 10:
        // some cool stuff
        break;
    case 42:
        // other stuff
        break;
    default:
        // fallback
}
```

Here, if `a` is either `2` or `10`, it will execute the "some cool stuff" code statements.

Another form of conditional in JavaScript is the "conditional operator," often called the "ternary operator." It's like a more concise form of a single `if..else` statement, such as:

```
var a = 42;

var b = (a > 41) ? "hello" : "world";

// similar to:

// if (a > 41) {
//    b = "hello";
// }
// else {
//    b = "world";
// }
```

If the test expression (`a > 41` here) evaluates as `true`, the first clause (`"hello"`) results, otherwise the second clause (`"world"`) results, and whatever the result is then gets assigned to `b`.

The conditional operator doesn't have to be used in an assignment, but that's definitely the most common usage.

Note: For more information about testing conditions and other patterns for `switch` and `? :`, see the *Types & Grammar* title of this series.

Strict Mode

ES5 added a "strict mode" to the language, which tightens the rules for certain behaviors. Generally, these restrictions are seen as keeping the code to a safer and more appropriate set of guidelines. Also, adhering to strict mode makes your code generally more optimizable by the engine. Strict mode is a big win for code, and you should use it for all your programs.

You can opt in to strict mode for an individual function, or an entire file, depending on where you put the strict mode pragma:

```
function foo() {
    "use strict";

    // this code is strict mode

    function bar() {
        // this code is strict mode
    }
}

// this code is not strict mode
```

Compare that to:

```
"use strict";

function foo() {
    // this code is strict mode

    function bar() {
        // this code is strict mode
    }
}

// this code is strict mode
```

One key difference (improvement!) with strict mode is disallowing the implicit auto-global variable declaration from omitting the `var` :

```
function foo() {
    "use strict";    // turn on strict mode
    a = 1;            // `var` missing, ReferenceError
}

foo();
```

If you turn on strict mode in your code, and you get errors, or code starts behaving buggy, your temptation might be to avoid strict mode. But that instinct would be a bad idea to indulge. If strict mode causes issues in your program, almost certainly it's a sign that you have things in your program you should fix.

Not only will strict mode keep your code to a safer path, and not only will it make your code more optimizable, but it also represents the future direction of the language. It'd be easier on you to get used to strict mode now than to keep putting it off -- it'll only get harder to convert later!

Note: For more information about strict mode, see the Chapter 5 of the *Types & Grammar* title of this series.

Functions As Values

So far, we've discussed functions as the primary mechanism of *scope* in JavaScript. You recall typical `function` declaration syntax as follows:

```
function foo() {
    // ..
}
```

Though it may not seem obvious from that syntax, `foo` is basically just a variable in the outer enclosing scope that's given a reference to the `function` being declared. That is, the `function` itself is a value, just like `42` or `[1,2,3]` would be.

This may sound like a strange concept at first, so take a moment to ponder it. Not only can you pass a value (argument) *to* a function, but *a function itself can be a value* that's assigned to variables, or passed to or returned from other functions.

As such, a function value should be thought of as an expression, much like any other value or expression.

Consider:

```
var foo = function() {
    // ..
};

var x = function bar(){
    // ..
};
```

The first function expression assigned to the `foo` variable is called *anonymous* because it has no `name`.

The second function expression is *named* (`bar`), even as a reference to it is also assigned to the `x` variable. *Named function expressions* are generally more preferable, though *anonymous function expressions* are still extremely common.

For more information, see the *Scope & Closures* title of this series.

Immediately Invoked Function Expressions (IIFEs)

In the previous snippet, neither of the function expressions are executed -- we could if we had included `foo()` or `x()`, for instance.

There's another way to execute a function expression, which is typically referred to as an *immediately invoked function expression* (IIFE):

```
(function IIFE(){
    console.log( "Hello!" );
})();
// "Hello!"
```

The outer `( .. )` that surrounds the `(function IIFE(){ .. })` function expression is just a nuance of JS grammar needed to prevent it from being treated as a normal function declaration.

The final `()` on the end of the expression -- the `})();` line -- is what actually executes the function expression referenced immediately before it.

That may seem strange, but it's not as foreign as first glance. Consider the similarities between `foo` and `IIFE` here:

```
function foo() { .. }

// `foo` function reference expression,
// then `()` executes it
foo();

// `IIFE` function expression,
// then `()` executes it
(function IIFE(){ .. })();
```

As you can see, listing the `(function IIFE(){ .. })` before its executing `()` is essentially the same as including `foo` before its executing `()`; in both cases, the function reference is executed with `()` immediately after it.

Because an IIFE is just a function, and functions create variable *scope*, using an IIFE in this fashion is often used to declare variables that won't affect the surrounding code outside the IIFE:

```
var a = 42;

(function IIFE(){
    var a = 10;
    console.log( a );    // 10
})();

console.log( a );        // 42
```

IIFEs can also have return values:

```
var x = (function IIFE(){
    return 42;
})();

x;    // 42
```

The `42` value gets `return` ed from the `IIFE` -named function being executed, and is then assigned to `x` .

Closure

Closure is one of the most important, and often least understood, concepts in JavaScript. I won't cover it in deep detail here, and instead refer you to the *Scope & Closures* title of this series. But I want to say a few things about it so you understand the general concept. It will be one of the most important techniques in your JS skillset.

You can think of closure as a way to "remember" and continue to access a function's scope (its variables) even once the function has finished running.

Consider:

```
function makeAdder(x) {
    // parameter `x` is an inner variable

    // inner function `add()` uses `x`, so
    // it has a "closure" over it
    function add(y) {
        return y + x;
    };

    return add;
}
```

The reference to the inner `add(..)` function that gets returned with each call to the outer `makeAdder(..)` is able to remember whatever `x` value was passed in to `makeAdder(..)`. Now, let's use `makeAdder(..)`:

```
// `plusOne` gets a reference to the inner `add(..)`
// function with closure over the `x` parameter of
// the outer `makeAdder(..)`
var plusOne = makeAdder( 1 );

// `plusTen` gets a reference to the inner `add(..)`
// function with closure over the `x` parameter of
// the outer `makeAdder(..)`
var plusTen = makeAdder( 10 );

plusOne( 3 );        // 4  <-- 1 + 3
plusOne( 41 );        // 42 <-- 1 + 41

plusTen( 13 );        // 23 <-- 10 + 13
```

More on how this code works:

1. When we call `makeAdder(1)`, we get back a reference to its inner `add(..)` that remembers `x` as `1`. We call this function reference `plusOne(..)`.
2. When we call `makeAdder(10)`, we get back another reference to its inner `add(..)` that remembers `x` as `10`. We call this function reference `plusTen(..)`.
3. When we call `plusOne(3)`, it adds `3` (its inner `y`) to the `1` (remembered by `x`), and we get `4` as the result.
4. When we call `plusTen(13)`, it adds `13` (its inner `y`) to the `10` (remembered by `x`), and we get `23` as the result.

Don't worry if this seems strange and confusing at first -- it can be! It'll take lots of practice to understand it fully.

But trust me, once you do, it's one of the most powerful and useful techniques in all of programming. It's definitely worth the effort to let your brain simmer on closures for a bit. In the next section, we'll get a little more practice with closure.

Modules

The most common usage of closure in JavaScript is the module pattern. Modules let you define private implementation details (variables, functions) that are hidden from the outside world, as well as a public API that *is* accessible from the outside.

Consider:

```
function User(){
    var username, password;

    function doLogin(user,pw) {
        username = user;
        password = pw;

        // do the rest of the login work
    }

    var publicAPI = {
        login: doLogin
    };

    return publicAPI;
}

// create a `User` module instance
var fred = User();

fred.login( "fred", "12Battery34!" );
```

The `User()` function serves as an outer scope that holds the variables `username` and `password`, as well as the inner `doLogin()` function; these are all private inner details of this `User` module that cannot be accessed from the outside world.

Warning: We are not calling `new User()` here, on purpose, despite the fact that probably seems more common to most readers. `User()` is just a function, not a class to be instantiated, so it's just called normally. Using `new` would be inappropriate and actually waste resources.

Executing `User()` creates an *instance* of the `User` module -- a whole new scope is created, and thus a whole new copy of each of these inner variables/functions. We assign this instance to `fred`. If we run `User()` again, we'd get a new instance entirely separate from `fred`.

The inner `doLogin()` function has a closure over `username` and `password`, meaning it will retain its access to them even after the `User()` function finishes running.

`publicAPI` is an object with one property/method on it, `login`, which is a reference to the inner `doLogin()` function. When we return `publicAPI` from `User()`, it becomes the instance we call `fred`.

At this point, the outer `User()` function has finished executing. Normally, you'd think the inner variables like `username` and `password` have gone away. But here they have not, because there's a closure in the `login()` function keeping them alive.

That's why we can call `fred.login(..)` -- the same as calling the inner `doLogin(..)` -- and it can still access `username` and `password` inner variables.

There's a good chance that with just this brief glimpse at closure and the module pattern, some of it is still a bit confusing. That's OK! It takes some work to wrap your brain around it.

From here, go read the *Scope & Closures* title of this series for a much more in-depth exploration.

`this` Identifier

Another very commonly misunderstood concept in JavaScript is the `this` identifier. Again, there's a couple of chapters on it in the *this & Object Prototypes* title of this series, so here we'll just briefly introduce the concept.

While it may often seem that `this` is related to "object-oriented patterns," in JS `this` is a different mechanism.

If a function has a `this` reference inside it, that `this` reference usually points to an `object`. But which `object` it points to depends on how the function was called.

It's important to realize that `this` *does not* refer to the function itself, as is the most common misconception.

Here's a quick illustration:

```
function foo() {
    console.log( this.bar );
}

var bar = "global";

var obj1 = {
    bar: "obj1",
    foo: foo
};

var obj2 = {
    bar: "obj2"
};

// --------

foo();                // "global"
obj1.foo();            // "obj1"
foo.call( obj2 );    // "obj2"
new foo();            // undefined
```

There are four rules for how `this` gets set, and they're shown in those last four lines of that snippet.

1. `foo()` ends up setting `this` to the global object in non-strict mode -- in strict mode, `this` would be `undefined` and you'd get an error in accessing the `bar` property -- so `"global"` is the value found for `this.bar` .
2. `obj1.foo()` sets `this` to the `obj1` object.
3. `foo.call(obj2)` sets `this` to the `obj2` object.
4. `new foo()` sets `this` to a brand new empty object.

Bottom line: to understand what `this` points to, you have to examine how the function in question was called. It will be one of those four ways just shown, and that will then answer what `this` is.

Note: For more information about `this` , see Chapters 1 and 2 of the *this & Object Prototypes* title of this series.

Prototypes

The prototype mechanism in JavaScript is quite complicated. We will only glance at it here. You will want to spend plenty of time reviewing Chapters 4-6 of the *this & Object Prototypes* title of this series for all the details.

When you reference a property on an object, if that property doesn't exist, JavaScript will automatically use that object's internal prototype reference to find another object to look for the property on. You could think of this almost as a fallback if the property is missing.

The internal prototype reference linkage from one object to its fallback happens at the time the object is created. The simplest way to illustrate it is with a built-in utility called `Object.create(..)` .

Consider:

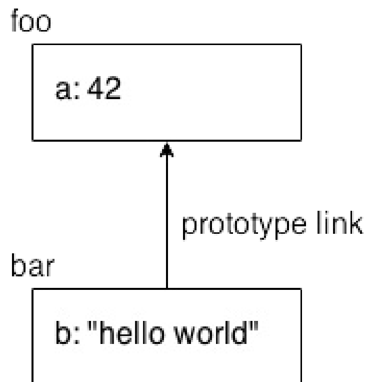
```
var foo = {
    a: 42
};

// create `bar` and link it to `foo`
var bar = Object.create( foo );

bar.b = "hello world";

bar.b;        // "hello world"
bar.a;        // 42 <-- delegated to `foo`
```

It may help to visualize the `foo` and `bar` objects and their relationship:



The `a` property doesn't actually exist on the `bar` object, but because `bar` is prototype-linked to `foo`, JavaScript automatically falls back to looking for `a` on the `foo` object, where it's found.

This linkage may seem like a strange feature of the language. The most common way this feature is used -- and I would argue, abused -- is to try to emulate/fake a "class" mechanism with "inheritance."

But a more natural way of applying prototypes is a pattern called "behavior delegation," where you intentionally design your linked objects to be able to *delegate* from one to the other for parts of the needed behavior.

Note: For more information about prototypes and behavior delegation, see Chapters 4-6 of the *this & Object Prototypes* title of this series.

Old & New

Some of the JS features we've already covered, and certainly many of the features covered in the rest of this series, are newer additions and will not necessarily be available in older browsers. In fact, some of the newest features in the specification aren't even implemented in any stable browsers yet.

So, what do you do with the new stuff? Do you just have to wait around for years or decades for all the old browsers to fade into obscurity?

That's how many people think about the situation, but it's really not a healthy approach to JS.

There are two main techniques you can use to "bring" the newer JavaScript stuff to the older browsers: polyfilling and transpiling.

Polyfilling

The word "polyfill" is an invented term (by Remy Sharp) (https://remysharp.com/2010/10/08/what-is-a-polyfill) used to refer to taking the definition of a newer feature and producing a piece of code that's equivalent to the behavior, but is able to run in older JS environments.

For example, ES6 defines a utility called `Number.isNaN(..)` to provide an accurate non-buggy check for `NaN` values, deprecating the original `isNaN(..)` utility. But it's easy to polyfill that utility so that you can start using it in your code regardless of whether the end user is in an ES6 browser or not.

Consider:

```
if (!Number.isNaN) {
    Number.isNaN = function isNaN(x) {
        return x !== x;
    };
}
```

The `if` statement guards against applying the polyfill definition in ES6 browsers where it will already exist. If it's not already present, we define `Number.isNaN(..)`.

Note: The check we do here takes advantage of a quirk with `NaN` values, which is that they're the only value in the whole language that is not equal to itself. So the `NaN` value is the only one that would make `x !== x` be `true`.

Not all new features are fully polyfillable. Sometimes most of the behavior can be polyfilled, but there are still small deviations. You should be really, really careful in implementing a polyfill yourself, to make sure you are adhering to the specification as strictly as possible.

Or better yet, use an already vetted set of polyfills that you can trust, such as those provided by ES5-Shim (https://github.com/es-shims/es5-shim) and ES6-Shim (https://github.com/es-shims/es6-shim).

Transpiling

There's no way to polyfill new syntax that has been added to the language. The new syntax would throw an error in the old JS engine as unrecognized/invalid.

So the better option is to use a tool that converts your newer code into older code equivalents. This process is commonly called "transpiling," a term for transforming + compiling.

Essentially, your source code is authored in the new syntax form, but what you deploy to the browser is the transpiled code in old syntax form. You typically insert the transpiler into your build process, similar to your code linter or your minifier.

You might wonder why you'd go to the trouble to write new syntax only to have it transpiled away to older code -- why not just write the older code directly?

There are several important reasons you should care about transpiling:

- The new syntax added to the language is designed to make your code more readable and maintainable. The older equivalents are often much more convoluted. You should prefer writing newer and cleaner syntax, not only for yourself but for all other members of the development team.
- If you transpile only for older browsers, but serve the new syntax to the newest browsers, you get to take advantage of browser performance optimizations with the new syntax. This also lets browser makers have more real-world code to test their implementations and optimizations on.
- Using the new syntax earlier allows it to be tested more robustly in the real world, which provides earlier feedback to the JavaScript committee (TC39). If issues are found early enough, they can be changed/fixed before those language design mistakes become permanent.

Here's a quick example of transpiling. ES6 adds a feature called "default parameter values." It looks like this:

```
function foo(a = 2) {
    console.log( a );
}

foo();          // 2
foo( 42 );      // 42
```

Simple, right? Helpful, too! But it's new syntax that's invalid in pre-ES6 engines. So what will a transpiler do with that code to make it run in older environments?

```
function foo() {
    var a = arguments[0] !== (void 0) ? arguments[0] : 2;
    console.log( a );
}
```

As you can see, it checks to see if the `arguments[0]` value is `void 0` (aka `undefined`), and if so provides the `2` default value; otherwise, it assigns whatever was passed.

In addition to being able to now use the nicer syntax even in older browsers, looking at the transpiled code actually explains the intended behavior more clearly.

You may not have realized just from looking at the ES6 version that `undefined` is the only value that can't get explicitly passed in for a default-value parameter, but the transpiled code makes that much more clear.

The last important detail to emphasize about transpilers is that they should now be thought of as a standard part of the JS development ecosystem and process. JS is going to continue to evolve, much more quickly than before, so every few months new syntax and new features will be added.

If you use a transpiler by default, you'll always be able to make that switch to newer syntax whenever you find it useful, rather than always waiting for years for today's browsers to phase out.

There are quite a few great transpilers for you to choose from. Here are some good options at the time of this writing:

- Babel (https://babeljs.io) (formerly 6to5): Transpiles ES6+ into ES5
- Traceur (https://github.com/google/traceur-compiler): Transpiles ES6, ES7, and beyond into ES5

Non-JavaScript

So far, the only things we've covered are in the JS language itself. The reality is that most JS is written to run in and interact with environments like browsers. A good chunk of the stuff that you write in your code is, strictly speaking, not directly controlled by JavaScript. That probably sounds a little strange.

The most common non-JavaScript JavaScript you'll encounter is the DOM API. For example:

```
var el = document.getElementById( "foo" );
```

The `document` variable exists as a global variable when your code is running in a browser. It's not provided by the JS engine, nor is it particularly controlled by the JavaScript specification. It takes the form of something that looks an awful lot like a normal JS `object` , but it's not really exactly that. It's a special `object,` often called a "host object."

Moreover, the `getElementById(..)` method on `document` looks like a normal JS function, but it's just a thinly exposed interface to a built-in method provided by the DOM from your browser. In some (newer-generation) browsers, this layer may also be in JS, but traditionally

the DOM and its behavior is implemented in something more like C/C++.

Another example is with input/output (I/O).

Everyone's favorite `alert(..)` pops up a message box in the user's browser window. `alert(..)` is provided to your JS program by the browser, not by the JS engine itself. The call you make sends the message to the browser internals and it handles drawing and displaying the message box.

The same goes with `console.log(..)`; your browser provides such mechanisms and hooks them up to the developer tools.

This book, and this whole series, focuses on JavaScript the language. That's why you don't see any substantial coverage of these non-JavaScript JavaScript mechanisms. Nevertheless, you need to be aware of them, as they'll be in every JS program you write!

Review

The first step to learning JavaScript's flavor of programming is to get a basic understanding of its core mechanisms like values, types, function closures, `this`, and prototypes.

Of course, each of these topics deserves much greater coverage than you've seen here, but that's why they have chapters and books dedicated to them throughout the rest of this series. After you feel pretty comfortable with the concepts and code samples in this chapter, the rest of the series awaits you to really dig in and get to know the language deeply.

The final chapter of this book will briefly summarize each of the other titles in the series and the other concepts they cover besides what we've already explored.

You Don't Know JS: Up & Going

Chapter 3: Into YDKJS

What is this series all about? Put simply, it's about taking seriously the task of learning *all parts of JavaScript*, not just some subset of the language that someone called "the good parts," and not just whatever minimal amount you need to get your job done at work.

Serious developers in other languages expect to put in the effort to learn most or all of the language(s) they primarily write in, but JS developers seem to stand out from the crowd in the sense of typically not learning very much of the language. This is not a good thing, and it's not something we should continue to allow to be the norm.

The *You Don't Know JS* (*YDKJS*) series stands in stark contrast to the typical approaches to learning JS, and is unlike almost any other JS books you will read. It challenges you to go beyond your comfort zone and to ask the deeper "why" questions for every single behavior you encounter. Are you up for that challenge?

I'm going to use this final chapter to briefly summarize what to expect from the rest of the books in the series, and how to most effectively go about building a foundation of JS learning on top of *YDKJS*.

Scope & Closures

Perhaps one of the most fundamental things you'll need to quickly come to terms with is how scoping of variables really works in JavaScript. It's not enough to have anecdotal fuzzy *beliefs* about scope.

The *Scope & Closures* title starts by debunking the common misconception that JS is an "interpreted language" and therefore not compiled. Nope.

The JS engine compiles your code right before (and sometimes during!) execution. So we use some deeper understanding of the compiler's approach to our code to understand how it finds and deals with variable and function declarations. Along the way, we see the typical metaphor for JS variable scope management, "Hoisting."

This critical understanding of "lexical scope" is what we then base our exploration of closure on for the last chapter of the book. Closure is perhaps the single most important concept in all of JS, but if you haven't first grasped firmly how scope works, closure will likely remain

beyond your grasp.

One important application of closure is the module pattern, as we briefly introduced in this book in Chapter 2. The module pattern is perhaps the most prevalent code organization pattern in all of JavaScript; deep understanding of it should be one of your highest priorities.

this & Object Prototypes

Perhaps one of the most widespread and persistent mistruths about JavaScript is that the `this` keyword refers to the function it appears in. Terribly mistaken.

The `this` keyword is dynamically bound based on how the function in question is executed, and it turns out there are four simple rules to understand and fully determine `this` binding.

Closely related to the `this` keyword is the object prototype mechanism, which is a look-up chain for properties, similar to how lexical scope variables are found. But wrapped up in the prototypes is the other huge miscue about JS: the idea of emulating (fake) classes and (so-called "prototypal") inheritance.

Unfortunately, the desire to bring class and inheritance design pattern thinking to JavaScript is just about the worst thing you could try to do, because while the syntax may trick you into thinking there's something like classes present, in fact the prototype mechanism is fundamentally opposite in its behavior.

What's at issue is whether it's better to ignore the mismatch and pretend that what you're implementing is "inheritance," or whether it's more appropriate to learn and embrace how the object prototype system actually works. The latter is more appropriately named "behavior delegation."

This is more than syntactic preference. Delegation is an entirely different, and more powerful, design pattern, one that replaces the need to design with classes and inheritance. But these assertions will absolutely fly in the face of nearly every other blog post, book, and conference talk on the subject for the entirety of JavaScript's lifetime.

The claims I make regarding delegation versus inheritance come not from a dislike of the language and its syntax, but from the desire to see the true capability of the language properly leveraged and the endless confusion and frustration wiped away.

But the case I make regarding prototypes and delegation is a much more involved one than what I will indulge here. If you're ready to reconsider everything you think you know about JavaScript "classes" and "inheritance," I offer you the chance to "take the red pill" (*Matrix* 1999) and check out Chapters 4-6 of the *this & Object Prototypes* title of this series.

Types & Grammar

The third title in this series primarily focuses on tackling yet another highly controversial topic: type coercion. Perhaps no topic causes more frustration with JS developers than when you talk about the confusions surrounding implicit coercion.

By far, the conventional wisdom is that implicit coercion is a "bad part" of the language and should be avoided at all costs. In fact, some have gone so far as to call it a "flaw" in the design of the language. Indeed, there are tools whose entire job is to do nothing but scan your code and complain if you're doing anything even remotely like coercion.

But is coercion really so confusing, so bad, so treacherous, that your code is doomed from the start if you use it?

I say no. After having built up an understanding of how types and values really work in Chapters 1-3, Chapter 4 takes on this debate and fully explains how coercion works, in all its nooks and crevices. We see just what parts of coercion really are surprising and what parts actually make complete sense if given the time to learn.

But I'm not merely suggesting that coercion is sensible and learnable, I'm asserting that coercion is an incredibly useful and totally underestimated tool that *you should be using in your code.* I'm saying that coercion, when used properly, not only works, but makes your code better. All the naysayers and doubters will surely scoff at such a position, but I believe it's one of the main keys to upping your JS game.

Do you want to just keep following what the crowd says, or are you willing to set all the assumptions aside and look at coercion with a fresh perspective? The *Types & Grammar* title of this series will coerce your thinking.

Async & Performance

The first three titles of this series focus on the core mechanics of the language, but the fourth title branches out slightly to cover patterns on top of the language mechanics for managing asynchronous programming. Asynchrony is not only critical to the performance of our applications, it's increasingly becoming *the* critical factor in writability and maintainability.

The book starts first by clearing up a lot of terminology and concept confusion around things like "async," "parallel," and "concurrent," and explains in depth how such things do and do not apply to JS.

Then we move into examining callbacks as the primary method of enabling asynchrony. But it's here that we quickly see that the callback alone is hopelessly insufficient for the modern demands of asynchronous programming. We identify two major deficiencies of callbacks-

only coding: *Inversion of Control* (IoC) trust loss and lack of linear reason-ability.

To address these two major deficiencies, ES6 introduces two new mechanisms (and indeed, patterns): promises and generators.

Promises are a time-independent wrapper around a "future value," which lets you reason about and compose them regardless of if the value is ready or not yet. Moreover, they effectively solve the IoC trust issues by routing callbacks through a trustable and composable promise mechanism.

Generators introduce a new mode of execution for JS functions, whereby the generator can be paused at `yield` points and be resumed asynchronously later. The pause-and-resume capability enables synchronous, sequential looking code in the generator to be processed asynchronously behind the scenes. By doing so, we address the non-linear, non-local-jump confusions of callbacks and thereby make our asynchronous code sync-looking so as to be more reason-able.

But it's the combination of promises and generators that "yields" our most effective asynchronous coding pattern to date in JavaScript. In fact, much of the future sophistication of asynchrony coming in ES7 and later will certainly be built on this foundation. To be serious about programming effectively in an async world, you're going to need to get really comfortable with combining promises and generators.

If promises and generators are about expressing patterns that let our programs run more concurrently and thus get more processing accomplished in a shorter period, JS has many other facets of performance optimization worth exploring.

Chapter 5 delves into topics like program parallelism with Web Workers and data parallelism with SIMD, as well as low-level optimization techniques like ASM.js. Chapter 6 takes a look at performance optimization from the perspective of proper benchmarking techniques, including what kinds of performance to worry about and what to ignore.

Writing JavaScript effectively means writing code that can break the constraint barriers of being run dynamically in a wide range of browsers and other environments. It requires a lot of intricate and detailed planning and effort on our parts to take a program from "it works" to "it works well."

The *Async & Performance* title is designed to give you all the tools and skills you need to write reasonable and performant JavaScript code.

ES6 & Beyond

No matter how much you feel you've mastered JavaScript to this point, the truth is that JavaScript is never going to stop evolving, and moreover, the rate of evolution is increasing rapidly. This fact is almost a metaphor for the spirit of this series, to embrace that we'll never fully *know* every part of JS, because as soon as you master it all, there's going to be new stuff coming down the line that you'll need to learn.

This title is dedicated to both the short- and mid-term visions of where the language is headed, not just the *known* stuff like ES6 but the *likely* stuff beyond.

While all the titles of this series embrace the state of JavaScript at the time of this writing, which is mid-way through ES6 adoption, the primary focus in the series has been more on ES5. Now, we want to turn our attention to ES6, ES7, and ...

Since ES6 is nearly complete at the time of this writing, *ES6 & Beyond* starts by dividing up the concrete stuff from the ES6 landscape into several key categories, including new syntax, new data structures (collections), and new processing capabilities and APIs. We cover each of these new ES6 features, in varying levels of detail, including reviewing details that are touched on in other books of this series.

Some exciting ES6 things to look forward to reading about: destructuring, default parameter values, symbols, concise methods, computed properties, arrow functions, block scoping, promises, generators, iterators, modules, proxies, weakmaps, and much, much more! Phew, ES6 packs quite a punch!

The first part of the book is a roadmap for all the stuff you need to learn to get ready for the new and improved JavaScript you'll be writing and exploring over the next couple of years.

The latter part of the book turns attention to briefly glance at things that we can likely expect to see in the near future of JavaScript. The most important realization here is that post-ES6, JS is likely going to evolve feature by feature rather than version by version, which means we can expect to see these near-future things coming much sooner than you might imagine.

The future for JavaScript is bright. Isn't it time we start learning it!?

Review

The *YDKJS* series is dedicated to the proposition that all JS developers can and should learn all of the parts of this great language. No person's opinion, no framework's assumptions, and no project's deadline should be the excuse for why you never learn and deeply understand JavaScript.

We take each important area of focus in the language and dedicate a short but very dense book to fully explore all the parts of it that you perhaps thought you knew but probably didn't fully.

"You Don't Know JS" isn't a criticism or an insult. It's a realization that all of us, myself included, must come to terms with. Learning JavaScript isn't an end goal but a process. We don't know JavaScript, yet. But we will!

You Don't Know JS: Up & Going

Appendix A: Acknowledgments

I have many people to thank for making this book title and the overall series happen.

First, I must thank my wife Christen Simpson, and my two kids Ethan and Emily, for putting up with Dad always pecking away at the computer. Even when not writing books, my obsession with JavaScript glues my eyes to the screen far more than it should. That time I borrow from my family is the reason these books can so deeply and completely explain JavaScript to you, the reader. I owe my family everything.

I'd like to thank my editors at O'Reilly, namely Simon St.Laurent and Brian MacDonald, as well as the rest of the editorial and marketing staff. They are fantastic to work with, and have been especially accommodating during this experiment into "open source" book writing, editing, and production.

Thank you to the many folks who have participated in making this book series better by providing editorial suggestions and corrections, including Shelley Powers, Tim Ferro, Evan Borden, Forrest L. Norvell, Jennifer Davis, Jesse Harlin, Kris Kowal, Rick Waldron, Jordan Harband, Benjamin Gruenbaum, Vyacheslav Egorov, David Nolen, and many others. A big thank you to Jenn Lukas for writing the Foreword for this title.

Thank you to the countless folks in the community, including members of the TC39 committee, who have shared so much knowledge with the rest of us, and especially tolerated my incessant questions and explorations with patience and detail. John-David Dalton, Juriy "kangax" Zaytsev, Mathias Bynens, Axel Rauschmayer, Nicholas Zakas, Angus Croll, Reginald Braithwaite, Dave Herman, Brendan Eich, Allen Wirfs-Brock, Bradley Meck, Domenic Denicola, David Walsh, Tim Disney, Peter van der Zee, Andrea Giammarchi, Kit Cambridge, Eric Elliott, and so many others, I can't even scratch the surface.

Since the "You Don't Know JS" book series was born on Kickstarter, I also wish to thank all my (nearly) 500 generous backers, without whom this book series could not have happened:

> Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark,

> Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsden, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel בר-לבב Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuț Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes,

> Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave ♥♬★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry,

> Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard

This book series is being produced in an open source fashion, including editing and production. We owe GitHub a debt of gratitude for making that sort of thing possible for the community!

Thank you again to all the countless folks I didn't name but to whom I nonetheless owe thanks. May this book series be "owned" by all of us and serve to contribute to increasing awareness and understanding of the JavaScript language, to the benefit of all current and future community contributors.

You Don't Know JS: Scope & Closures

Chapter 1: What is Scope?

One of the most fundamental paradigms of nearly all programming languages is the ability to store values in variables, and later retrieve or modify those values. In fact, the ability to store values and pull values out of variables is what gives a program *state*.

Without such a concept, a program could perform some tasks, but they would be extremely limited and not terribly interesting.

But the inclusion of variables into our program begets the most interesting questions we will now address: where do those variables *live*? In other words, where are they stored? And, most importantly, how does our program find them when it needs them?

These questions speak to the need for a well-defined set of rules for storing variables in some location, and for finding those variables at a later time. We'll call that set of rules: *Scope*.

But, where and how do these *Scope* rules get set?

Compiler Theory

It may be self-evident, or it may be surprising, depending on your level of interaction with various languages, but despite the fact that JavaScript falls under the general category of "dynamic" or "interpreted" languages, it is in fact a compiled language. It is *not* compiled well in advance, as are many traditionally-compiled languages, nor are the results of compilation portable among various distributed systems.

But, nevertheless, the JavaScript engine performs many of the same steps, albeit in more sophisticated ways than we may commonly be aware, of any traditional language-compiler.

In traditional compiled-language process, a chunk of source code, your program, will undergo typically three steps *before* it is executed, roughly called "compilation":

1. **Tokenizing/Lexing:** breaking up a string of characters into meaningful (to the language) chunks, called tokens. For instance, consider the program: `var a = 2;`. This program would likely be broken up into the following tokens: `var`, `a`, `=`, `2`, and `;`. Whitespace may or may not be persisted as a token, depending on whether it's meaningful or not.

> **Note:** The difference between tokenizing and lexing is subtle and academic, but it centers on whether or not these tokens are identified in a *stateless* or *stateful* way. Put simply, if the tokenizer were to invoke stateful parsing rules to figure out whether `a` should be considered a distinct token or just part of another token, *that* would be **lexing**.

2. **Parsing:** taking a stream (array) of tokens and turning it into a tree of nested elements, which collectively represent the grammatical structure of the program. This tree is called an "AST" (**A**bstract **S**yntax **T**ree).

   The tree for `var a = 2;` might start with a top-level node called `VariableDeclaration`, with a child node called `Identifier` (whose value is `a`), and another child called `AssignmentExpression` which itself has a child called `NumericLiteral` (whose value is `2`).

3. **Code-Generation:** the process of taking an AST and turning it into executable code. This part varies greatly depending on the language, the platform it's targeting, etc.

   So, rather than get mired in details, we'll just handwave and say that there's a way to take our above described AST for `var a = 2;` and turn it into a set of machine instructions to actually *create* a variable called `a` (including reserving memory, etc.), and then store a value into `a`.

   > **Note:** The details of how the engine manages system resources are deeper than we will dig, so we'll just take it for granted that the engine is able to create and store variables as needed.

The JavaScript engine is vastly more complex than *just* those three steps, as are most other language compilers. For instance, in the process of parsing and code-generation, there are certainly steps to optimize the performance of the execution, including collapsing redundant elements, etc.

So, I'm painting only with broad strokes here. But I think you'll see shortly why *these* details we *do* cover, even at a high level, are relevant.

For one thing, JavaScript engines don't get the luxury (like other language compilers) of having plenty of time to optimize, because JavaScript compilation doesn't happen in a build step ahead of time, as with other languages.

For JavaScript, the compilation that occurs happens, in many cases, mere microseconds (or less!) before the code is executed. To ensure the fastest performance, JS engines use all kinds of tricks (like JITs, which lazy compile and even hot re-compile, etc.) which are well beyond the "scope" of our discussion here.

Let's just say, for simplicity's sake, that any snippet of JavaScript has to be compiled before (usually *right* before!) it's executed. So, the JS compiler will take the program `var a = 2;` and compile it *first*, and then be ready to execute it, usually right away.

Understanding Scope

The way we will approach learning about scope is to think of the process in terms of a conversation. But, *who* is having the conversation?

The Cast

Let's meet the cast of characters that interact to process the program `var a = 2;` , so we understand their conversations that we'll listen in on shortly:

1. *Engine*: responsible for start-to-finish compilation and execution of our JavaScript program.

2. *Compiler*: one of *Engine*'s friends; handles all the dirty work of parsing and code-generation (see previous section).

3. *Scope*: another friend of *Engine*; collects and maintains a look-up list of all the declared identifiers (variables), and enforces a strict set of rules as to how these are accessible to currently executing code.

For you to *fully understand* how JavaScript works, you need to begin to *think* like *Engine* (and friends) think, ask the questions they ask, and answer those questions the same.

Back & Forth

When you see the program `var a = 2;` , you most likely think of that as one statement. But that's not how our new friend *Engine* sees it. In fact, *Engine* sees two distinct statements, one which *Compiler* will handle during compilation, and one which *Engine* will handle during execution.

So, let's break down how *Engine* and friends will approach the program `var a = 2;` .

The first thing *Compiler* will do with this program is perform lexing to break it down into tokens, which it will then parse into a tree. But when *Compiler* gets to code-generation, it will treat this program somewhat differently than perhaps assumed.

A reasonable assumption would be that *Compiler* will produce code that could be summed up by this pseudo-code: "Allocate memory for a variable, label it `a` , then stick the value `2` into that variable." Unfortunately, that's not quite accurate.

Compiler will instead proceed as:

1. Encountering `var a`, *Compiler* asks *Scope* to see if a variable `a` already exists for that particular scope collection. If so, *Compiler* ignores this declaration and moves on. Otherwise, *Compiler* asks *Scope* to declare a new variable called `a` for that scope collection.

2. *Compiler* then produces code for *Engine* to later execute, to handle the `a = 2` assignment. The code *Engine* runs will first ask *Scope* if there is a variable called `a` accessible in the current scope collection. If so, *Engine* uses that variable. If not, *Engine* looks *elsewhere* (see nested *Scope* section below).

If *Engine* eventually finds a variable, it assigns the value `2` to it. If not, *Engine* will raise its hand and yell out an error!

To summarize: two distinct actions are taken for a variable assignment: First, *Compiler* declares a variable (if not previously declared in the current scope), and second, when executing, *Engine* looks up the variable in *Scope* and assigns to it, if found.

Compiler Speak

We need a little bit more compiler terminology to proceed further with understanding.

When *Engine* executes the code that *Compiler* produced for step (2), it has to look-up the variable `a` to see if it has been declared, and this look-up is consulting *Scope*. But the type of look-up *Engine* performs affects the outcome of the look-up.

In our case, it is said that *Engine* would be performing an "LHS" look-up for the variable `a`. The other type of look-up is called "RHS".

I bet you can guess what the "L" and "R" mean. These terms stand for "Left-hand Side" and "Right-hand Side".

Side... of what? **Of an assignment operation.**

In other words, an LHS look-up is done when a variable appears on the left-hand side of an assignment operation, and an RHS look-up is done when a variable appears on the right-hand side of an assignment operation.

Actually, let's be a little more precise. An RHS look-up is indistinguishable, for our purposes, from simply a look-up of the value of some variable, whereas the LHS look-up is trying to find the variable container itself, so that it can assign. In this way, RHS doesn't *really* mean "right-hand side of an assignment" per se, it just, more accurately, means "not left-hand side".

Being slightly glib for a moment, you could also think "RHS" instead means "retrieve his/her source (value)", implying that RHS means "go get the value of...".

Let's dig into that deeper.

When I say:

```
console.log( a );
```

The reference to `a` is an RHS reference, because nothing is being assigned to `a` here. Instead, we're looking-up to retrieve the value of `a`, so that the value can be passed to `console.log(..)`.

By contrast:

```
a = 2;
```

The reference to `a` here is an LHS reference, because we don't actually care what the current value is, we simply want to find the variable as a target for the `= 2` assignment operation.

Note: LHS and RHS meaning "left/right-hand side of an assignment" doesn't necessarily literally mean "left/right side of the `=` assignment operator". There are several other ways that assignments happen, and so it's better to conceptually think about it as: "who's the target of the assignment (LHS)" and "who's the source of the assignment (RHS)".

Consider this program, which has both LHS and RHS references:

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

The last line that invokes `foo(..)` as a function call requires an RHS reference to `foo`, meaning, "go look-up the value of `foo`, and give it to me." Moreover, `(..)` means the value of `foo` should be executed, so it'd better actually be a function!

There's a subtle but important assignment here. **Did you spot it?**

You may have missed the implied `a = 2` in this code snippet. It happens when the value `2` is passed as an argument to the `foo(..)` function, in which case the `2` value is **assigned** to the parameter `a`. To (implicitly) assign to parameter `a`, an LHS look-up is performed.

There's also an RHS reference for the value of `a`, and that resulting value is passed to `console.log(..)`. `console.log(..)` needs a reference to execute. It's an RHS look-up for the `console` object, then a property-resolution occurs to see if it has a method called `log`.

Finally, we can conceptualize that there's an LHS/RHS exchange of passing the value `2` (by way of variable `a`'s RHS look-up) into `log(..)`. Inside of the native implementation of `log(..)`, we can assume it has parameters, the first of which (perhaps called `arg1`) has an LHS reference look-up, before assigning `2` to it.

Note: You might be tempted to conceptualize the function declaration `function foo(a) {...` as a normal variable declaration and assignment, such as `var foo` and `foo = function(a) {...`. In so doing, it would be tempting to think of this function declaration as involving an LHS look-up.

However, the subtle but important difference is that *Compiler* handles both the declaration and the value definition during code-generation, such that when *Engine* is executing code, there's no processing necessary to "assign" a function value to `foo`. Thus, it's not really appropriate to think of a function declaration as an LHS look-up assignment in the way we're discussing them here.

Engine/Scope Conversation

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

Let's imagine the above exchange (which processes this code snippet) as a conversation. The conversation would go a little something like this:

> ***Engine***: Hey *Scope*, I have an RHS reference for `foo`. Ever heard of it?
>
> ***Scope***: Why yes, I have. *Compiler* declared it just a second ago. He's a function. Here you go.
>
> ***Engine***: Great, thanks! OK, I'm executing `foo`.
>
> ***Engine***: Hey, *Scope*, I've got an LHS reference for `a`, ever heard of it?
>
> ***Scope***: Why yes, I have. *Compiler* declared it as a formal parameter to `foo` just recently. Here you go.
>
> ***Engine***: Helpful as always, *Scope*. Thanks again. Now, time to assign `2` to `a`.
>
> ***Engine***: Hey, *Scope*, sorry to bother you again. I need an RHS look-up for `console`. Ever heard of it?
>
> ***Scope***: No problem, *Engine*, this is what I do all day. Yes, I've got `console`. He's built-in. Here ya go.
>
> ***Engine***: Perfect. Looking up `log(..)`. OK, great, it's a function.
>
> ***Engine***: Yo, *Scope*. Can you help me out with an RHS reference to `a`. I think I remember it, but just want to double-check.
>
> ***Scope***: You're right, *Engine*. Same guy, hasn't changed. Here ya go.
>
> ***Engine***: Cool. Passing the value of `a`, which is `2`, into `log(..)`.
>
> ...

Quiz

Check your understanding so far. Make sure to play the part of *Engine* and have a "conversation" with the *Scope*:

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Identify all the LHS look-ups (there are 3!).
2. Identify all the RHS look-ups (there are 4!).

Note: See the chapter review for the quiz answers!

Nested Scope

We said that *Scope* is a set of rules for looking up variables by their identifier name. There's usually more than one *Scope* to consider, however.

Just as a block or function is nested inside another block or function, scopes are nested inside other scopes. So, if a variable cannot be found in the immediate scope, *Engine* consults the next outer containing scope, continuing until found or until the outermost (aka, global) scope has been reached.

Consider:

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

The RHS reference for `b` cannot be resolved inside the function `foo`, but it can be resolved in the *Scope* surrounding it (in this case, the global).

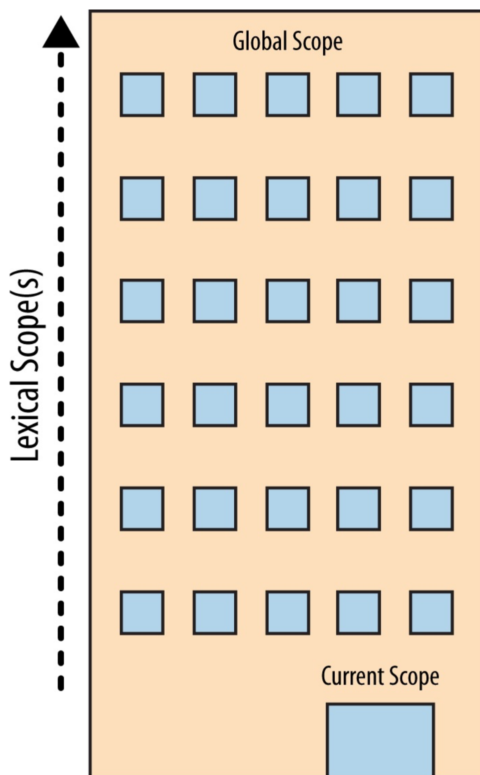
So, revisiting the conversations between *Engine* and *Scope*, we'd overhear:

> ***Engine***: "Hey, *Scope* of `foo`, ever heard of `b`? Got an RHS reference for it."
>
> ***Scope***: "Nope, never heard of it. Go fish."
>
> ***Engine***: "Hey, *Scope* outside of `foo`, oh you're the global *Scope*, ok cool. Ever heard of `b`? Got an RHS reference for it."
>
> ***Scope***: "Yep, sure have. Here ya go."

The simple rules for traversing nested *Scope*: *Engine* starts at the currently executing *Scope*, looks for the variable there, then if not found, keeps going up one level, and so on. If the outermost global scope is reached, the search stops, whether it finds the variable or not.

Building on Metaphors

To visualize the process of nested *Scope* resolution, I want you to think of this tall building.



The building represents our program's nested *Scope* rule set. The first floor of the building represents your currently executing *Scope*, wherever you are. The top level of the building is the global *Scope*.

You resolve LHS and RHS references by looking on your current floor, and if you don't find it, taking the elevator to the next floor, looking there, then the next, and so on. Once you get to the top floor (the global *Scope*), you either find what you're looking for, or you don't. But you have to stop regardless.

Errors

Why does it matter whether we call it LHS or RHS?

Because these two types of look-ups behave differently in the circumstance where the variable has not yet been declared (is not found in any consulted *Scope*).

Consider:

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

When the RHS look-up occurs for `b` the first time, it will not be found. This is said to be an "undeclared" variable, because it is not found in the scope.

If an RHS look-up fails to ever find a variable, anywhere in the nested *Scope*s, this results in a `ReferenceError` being thrown by the *Engine*. It's important to note that the error is of the type `ReferenceError`.

By contrast, if the *Engine* is performing an LHS look-up and arrives at the top floor (global *Scope*) without finding it, and if the program is not running in "Strict Mode" [note-strictmode], then the global *Scope* will create a new variable of that name **in the global scope**, and hand it back to *Engine*.

"No, there wasn't one before, but I was helpful and created one for you."

"Strict Mode" [note-strictmode], which was added in ES5, has a number of different behaviors from normal/relaxed/lazy mode. One such behavior is that it disallows the automatic/implicit global variable creation. In that case, there would be no global *Scope*'d variable to hand back from an LHS look-up, and *Engine* would throw a `ReferenceError` similarly to the RHS case.

Now, if a variable is found for an RHS look-up, but you try to do something with its value that is impossible, such as trying to execute-as-function a non-function value, or reference a property on a `null` or `undefined` value, then *Engine* throws a different kind of error, called a `TypeError`.

`ReferenceError` is *Scope* resolution-failure related, whereas `TypeError` implies that *Scope* resolution was successful, but that there was an illegal/impossible action attempted against the result.

Review (TL;DR)

Scope is the set of rules that determines where and how a variable (identifier) can be looked-up. This look-up may be for the purposes of assigning to the variable, which is an LHS (left-hand-side) reference, or it may be for the purposes of retrieving its value, which is an RHS (right-hand-side) reference.

LHS references result from assignment operations. *Scope*-related assignments can occur either with the `=` operator or by passing arguments to (assign to) function parameters.

The JavaScript *Engine* first compiles code before it executes, and in so doing, it splits up statements like `var a = 2;` into two separate steps:

1. First, `var a` to declare it in that *Scope*. This is performed at the beginning, before code execution.

2. Later, `a = 2` to look up the variable (LHS reference) and assign to it if found.

Both LHS and RHS reference look-ups start at the currently executing *Scope*, and if need be (that is, they don't find what they're looking for there), they work their way up the nested *Scope*, one scope (floor) at a time, looking for the identifier, until they get to the global (top floor) and stop, and either find it, or don't.

Unfulfilled RHS references result in `ReferenceError`s being thrown. Unfulfilled LHS references result in an automatic, implicitly-created global of that name (if not in "Strict Mode" [note-strictmode]), or a `ReferenceError` (if in "Strict Mode" [note-strictmode]).

Quiz Answers

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Identify all the LHS look-ups (there are 3!).

   `c = ..`, `a = 2` (implicit param assignment) and `b = ..`

2. Identify all the RHS look-ups (there are 4!).

   `foo(2..`, `= a;`, `a + ..` and `.. + b`

> note-strictmode. MDN: Strict Mode ↩

You Don't Know JS: Scope & Closures

Chapter 2: Lexical Scope

In Chapter 1, we defined "scope" as the set of rules that govern how the *Engine* can look up a variable by its identifier name and find it, either in the current *Scope*, or in any of the *Nested Scopes* it's contained within.

There are two predominant models for how scope works. The first of these is by far the most common, used by the vast majority of programming languages. It's called **Lexical Scope**, and we will examine it in-depth. The other model, which is still used by some languages (such as Bash scripting, some modes in Perl, etc.) is called **Dynamic Scope**.

Dynamic Scope is covered in Appendix A. I mention it here only to provide a contrast with Lexical Scope, which is the scope model that JavaScript employs.

Lex-time

As we discussed in Chapter 1, the first traditional phase of a standard language compiler is called lexing (aka, tokenizing). If you recall, the lexing process examines a string of source code characters and assigns semantic meaning to the tokens as a result of some stateful parsing.

It is this concept which provides the foundation to understand what lexical scope is and where the name comes from.

To define it somewhat circularly, lexical scope is scope that is defined at lexing time. In other words, lexical scope is based on where variables and blocks of scope are authored, by you, at write time, and thus is (mostly) set in stone by the time the lexer processes your code.

Note: We will see in a little bit there are some ways to cheat lexical scope, thereby modifying it after the lexer has passed by, but these are frowned upon. It is considered best practice to treat lexical scope as, in fact, lexical-only, and thus entirely author-time in nature.

Let's consider this block of code:

```
function foo(a) {

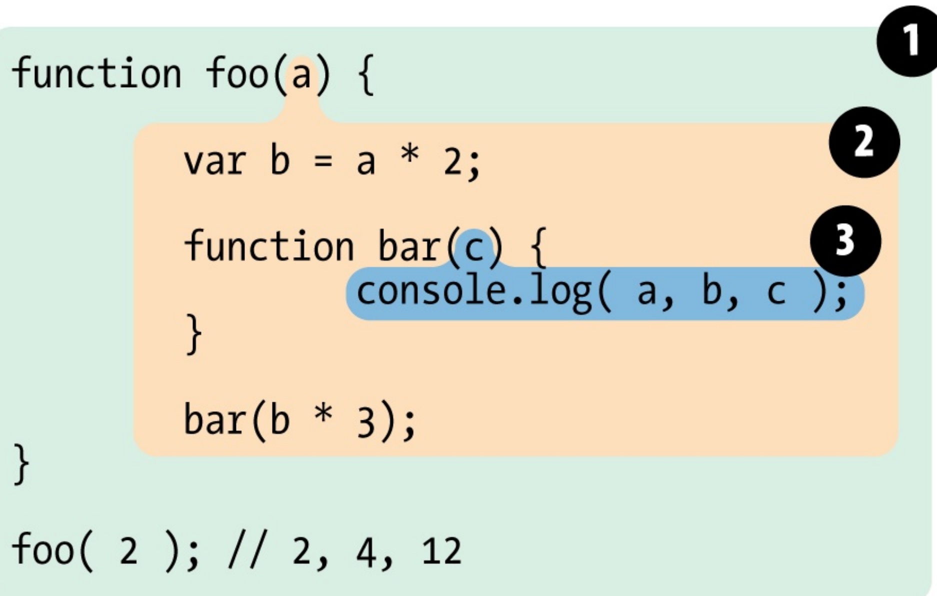
    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }

    bar(b * 3);
}

foo( 2 ); // 2 4 12
```

There are three nested scopes inherent in this code example. It may be helpful to think about these scopes as bubbles inside of each other.



Bubble 1 encompasses the global scope, and has just one identifier in it: `foo` .

Bubble 2 encompasses the scope of `foo` , which includes the three identifiers: `a` , `bar` and `b` .

Bubble 3 encompasses the scope of `bar` , and it includes just one identifier: `c` .

Scope bubbles are defined by where the blocks of scope are written, which one is nested inside the other, etc. In the next chapter, we'll discuss different units of scope, but for now, let's just assume that each function creates a new bubble of scope.

The bubble for `bar` is entirely contained within the bubble for `foo` , because (and only because) that's where we chose to define the function `bar` .

Notice that these nested bubbles are strictly nested. We're not talking about Venn diagrams where the bubbles can cross boundaries. In other words, no bubble for some function can simultaneously exist (partially) inside two other outer scope bubbles, just as no function can partially be inside each of two parent functions.

Look-ups

The structure and relative placement of these scope bubbles fully explains to the *Engine* all the places it needs to look to find an identifier.

In the above code snippet, the *Engine* executes the `console.log(..)` statement and goes looking for the three referenced variables `a` , `b` , and `c` . It first starts with the innermost scope bubble, the scope of the `bar(..)` function. It won't find `a` there, so it goes up one level, out to the next nearest scope bubble, the scope of `foo(..)` . It finds `a` there, and so it uses that `a` . Same thing for `b` . But `c` , it does find inside of `bar(..)` .

Had there been a `c` both inside of `bar(..)` and inside of `foo(..)` , the `console.log(..)` statement would have found and used the one in `bar(..)` , never getting to the one in `foo(..)` .

Scope look-up stops once it finds the first match. The same identifier name can be specified at multiple layers of nested scope, which is called "shadowing" (the inner identifier "shadows" the outer identifier). Regardless of shadowing, scope look-up always starts at the innermost scope being executed at the time, and works its way outward/upward until the first match, and stops.

Note: Global variables are also automatically properties of the global object (`window` in browsers, etc.), so it *is* possible to reference a global variable not directly by its lexical name, but instead indirectly as a property reference of the global object.

```
window.a
```

This technique gives access to a global variable which would otherwise be inaccessible due to it being shadowed. However, non-global shadowed variables cannot be accessed.

No matter *where* a function is invoked from, or even *how* it is invoked, its lexical scope is **only** defined by where the function was declared.

The lexical scope look-up process *only* applies to first-class identifiers, such as the `a` , `b` , and `c` . If you had a reference to `foo.bar.baz` in a piece of code, the lexical scope look-up would apply to finding the `foo` identifier, but once it locates that variable, object property-access rules take over to resolve the `bar` and `baz` properties, respectively.

Cheating Lexical

If lexical scope is defined only by where a function is declared, which is entirely an author-time decision, how could there possibly be a way to "modify" (aka, cheat) lexical scope at run-time?

JavaScript has two such mechanisms. Both of them are equally frowned-upon in the wider community as bad practices to use in your code. But the typical arguments against them are often missing the most important point: **cheating lexical scope leads to poorer performance.**

Before I explain the performance issue, though, let's look at how these two mechanisms work.

`eval`

The `eval(..)` function in JavaScript takes a string as an argument, and treats the contents of the string as if it had actually been authored code at that point in the program. In other words, you can programmatically generate code inside of your authored code, and run the generated code as if it had been there at author time.

Evaluating `eval(..)` (pun intended) in that light, it should be clear how `eval(..)` allows you to modify the lexical scope environment by cheating and pretending that author-time (aka, lexical) code was there all along.

On subsequent lines of code after an `eval(..)` has executed, the *Engine* will not "know" or "care" that the previous code in question was dynamically interpreted and thus modified the lexical scope environment. The *Engine* will simply perform its lexical scope look-ups as it always does.

Consider the following code:

```
function foo(str, a) {
    eval( str ); // cheating!
    console.log( a, b );
}

var b = 2;

foo( "var b = 3;", 1 ); // 1, 3
```

The string `"var b = 3;"` is treated, at the point of the `eval(..)` call, as code that was there all along. Because that code happens to declare a new variable `b`, it modifies the existing lexical scope of `foo(..)`. In fact, as mentioned above, this code actually creates variable

`b` inside of `foo(..)` that shadows the `b` that was declared in the outer (global) scope.

When the `console.log(..)` call occurs, it finds both `a` and `b` in the scope of `foo(..)`, and never finds the outer `b`. Thus, we print out "1, 3" instead of "1, 2" as would have normally been the case.

Note: In this example, for simplicity's sake, the string of "code" we pass in was a fixed literal. But it could easily have been programmatically created by adding characters together based on your program's logic. `eval(..)` is usually used to execute dynamically created code, as dynamically evaluating essentially static code from a string literal would provide no real benefit to just authoring the code directly.

By default, if a string of code that `eval(..)` executes contains one or more declarations (either variables or functions), this action modifies the existing lexical scope in which the `eval(..)` resides. Technically, `eval(..)` can be invoked "indirectly", through various tricks (beyond our discussion here), which causes it to instead execute in the context of the global scope, thus modifying it. But in either case, `eval(..)` can at runtime modify an author-time lexical scope.

Note: `eval(..)` when used in a strict-mode program operates in its own lexical scope, which means declarations made inside of the `eval()` do not actually modify the enclosing scope.

```
function foo(str) {
   "use strict";
   eval( str );
   console.log( a ); // ReferenceError: a is not defined
}

foo( "var a = 2" );
```

There are other facilities in JavaScript which amount to a very similar effect to `eval(..)`. `setTimeout(..)` and `setInterval(..)` *can* take a string for their respective first argument, the contents of which are `eval`uated as the code of a dynamically-generated function. This is old, legacy behavior and long-since deprecated. Don't do it!

The `new Function(..)` function constructor similarly takes a string of code in its **last** argument to turn into a dynamically-generated function (the first argument(s), if any, are the named parameters for the new function). This function-constructor syntax is slightly safer than `eval(..)`, but it should still be avoided in your code.

The use-cases for dynamically generating code inside your program are incredibly rare, as the performance degradations are almost never worth the capability.

`with`

The other frowned-upon (and now deprecated!) feature in JavaScript which cheats lexical scope is the `with` keyword. There are multiple valid ways that `with` can be explained, but I will choose here to explain it from the perspective of how it interacts with and affects lexical scope.

`with` is typically explained as a short-hand for making multiple property references against an object *without* repeating the object reference itself each time.

For example:

```
var obj = {
    a: 1,
    b: 2,
    c: 3
};

// more "tedious" to repeat "obj"
obj.a = 2;
obj.b = 3;
obj.c = 4;

// "easier" short-hand
with (obj) {
    a = 3;
    b = 4;
    c = 5;
}
```

However, there's much more going on here than just a convenient short-hand for object property access. Consider:

```
function foo(obj) {
    with (obj) {
        a = 2;
    }
}

var o1 = {
    a: 3
};

var o2 = {
    b: 3
};

foo( o1 );
console.log( o1.a ); // 2

foo( o2 );
console.log( o2.a ); // undefined
console.log( a ); // 2 -- Oops, leaked global!
```

In this code example, two objects `o1` and `o2` are created. One has an `a` property, and the other does not. The `foo(..)` function takes an object reference `obj` as an argument, and calls `with (obj) { .. }` on the reference. Inside the `with` block, we make what appears to be a normal lexical reference to a variable `a`, an LHS reference in fact (see Chapter 1), to assign to it the value of `2`.

When we pass in `o1`, the `a = 2` assignment finds the property `o1.a` and assigns it the value `2`, as reflected in the subsequent `console.log(o1.a)` statement. However, when we pass in `o2`, since it does not have an `a` property, no such property is created, and `o2.a` remains `undefined`.

But then we note a peculiar side-effect, the fact that a global variable `a` was created by the `a = 2` assignment. How can this be?

The `with` statement takes an object, one which has zero or more properties, and **treats that object as if *it* is a wholly separate lexical scope**, and thus the object's properties are treated as lexically defined identifiers in that "scope".

Note: Even though a `with` block treats an object like a lexical scope, a normal `var` declaration inside that `with` block will not be scoped to that `with` block, but instead the containing function scope.

While the `eval(..)` function can modify existing lexical scope if it takes a string of code with one or more declarations in it, the `with` statement actually creates a **whole new lexical scope** out of thin air, from the object you pass to it.

Understood in this way, the "scope" declared by the `with` statement when we passed in `o1` was `o1`, and that "scope" had an "identifier" in it which corresponds to the `o1.a` property. But when we used `o2` as the "scope", it had no such `a` "identifier" in it, and so the normal rules of LHS identifier look-up (see Chapter 1) occurred.

Neither the "scope" of `o2`, nor the scope of `foo(..)`, nor the global scope even, has an `a` identifier to be found, so when `a = 2` is executed, it results in the automatic-global being created (since we're in non-strict mode).

It is a strange sort of mind-bending thought to see `with` turning, at runtime, an object and its properties into a "scope" *with* "identifiers". But that is the clearest explanation I can give for the results we see.

Note: In addition to being a bad idea to use, both `eval(..)` and `with` are affected (restricted) by Strict Mode. `with` is outright disallowed, whereas various forms of indirect or unsafe `eval(..)` are disallowed while retaining the core functionality.

Performance

Both `eval(..)` and `with` cheat the otherwise author-time defined lexical scope by modifying or creating new lexical scope at runtime.

So, what's the big deal, you ask? If they offer more sophisticated functionality and coding flexibility, aren't these *good* features? **No.**

The JavaScript *Engine* has a number of performance optimizations that it performs during the compilation phase. Some of these boil down to being able to essentially statically analyze the code as it lexes, and pre-determine where all the variable and function declarations are, so that it takes less effort to resolve identifiers during execution.

But if the *Engine* finds an `eval(..)` or `with` in the code, it essentially has to *assume* that all its awareness of identifier location may be invalid, because it cannot know at lexing time exactly what code you may pass to `eval(..)` to modify the lexical scope, or the contents of the object you may pass to `with` to create a new lexical scope to be consulted.

In other words, in the pessimistic sense, most of those optimizations it *would* make are pointless if `eval(..)` or `with` are present, so it simply doesn't perform the optimizations *at all*.

Your code will almost certainly tend to run slower simply by the fact that you include an `eval(..)` or `with` anywhere in the code. No matter how smart the *Engine* may be about trying to limit the side-effects of these pessimistic assumptions, **there's no getting around the fact that without the optimizations, code runs slower.**

Review (TL;DR)

Lexical scope means that scope is defined by author-time decisions of where functions are declared. The lexing phase of compilation is essentially able to know where and how all identifiers are declared, and thus predict how they will be looked-up during execution.

Two mechanisms in JavaScript can "cheat" lexical scope: `eval(..)` and `with`. The former can modify existing lexical scope (at runtime) by evaluating a string of "code" which has one or more declarations in it. The latter essentially creates a whole new lexical scope (again, at runtime) by treating an object reference *as* a "scope" and that object's properties as scoped identifiers.

The downside to these mechanisms is that it defeats the *Engine*'s ability to perform compile-time optimizations regarding scope look-up, because the *Engine* has to assume pessimistically that such optimizations will be invalid. Code *will* run slower as a result of using either feature. **Don't use them.**

You Don't Know JS: Scope & Closures

Chapter 3: Function vs. Block Scope

As we explored in Chapter 2, scope consists of a series of "bubbles" that each act as a container or bucket, in which identifiers (variables, functions) are declared. These bubbles nest neatly inside each other, and this nesting is defined at author-time.

But what exactly makes a new bubble? Is it only the function? Can other structures in JavaScript create bubbles of scope?

Scope From Functions

The most common answer to those questions is that JavaScript has function-based scope. That is, each function you declare creates a bubble for itself, but no other structures create their own scope bubbles. As we'll see in just a little bit, this is not quite true.

But first, let's explore function scope and its implications.

Consider this code:

```
function foo(a) {
    var b = 2;

    // some code

    function bar() {
        // ...
    }

    // more code

    var c = 3;
}
```

In this snippet, the scope bubble for `foo(..)` includes identifiers `a`, `b`, `c` and `bar`. **It doesn't matter** *where* in the scope a declaration appears, the variable or function belongs to the containing scope bubble, regardless. We'll explore how exactly *that* works in the next chapter.

`bar(..)` has its own scope bubble. So does the global scope, which has just one identifier attached to it: `foo` .

Because `a` , `b` , `c` , and `bar` all belong to the scope bubble of `foo(..)` , they are not accessible outside of `foo(..)` . That is, the following code would all result in `ReferenceError` errors, as the identifiers are not available to the global scope:

```
bar(); // fails

console.log( a, b, c ); // all 3 fail
```

However, all these identifiers (`a` , `b` , `c` , `foo` , and `bar`) are accessible *inside* of `foo(..)` , and indeed also available inside of `bar(..)` (assuming there are no shadow identifier declarations inside `bar(..)`).

Function scope encourages the idea that all variables belong to the function, and can be used and reused throughout the entirety of the function (and indeed, accessible even to nested scopes). This design approach can be quite useful, and certainly can make full use of the "dynamic" nature of JavaScript variables to take on values of different types as needed.

On the other hand, if you don't take careful precautions, variables existing across the entirety of a scope can lead to some unexpected pitfalls.

Hiding In Plain Scope

The traditional way of thinking about functions is that you declare a function, and then add code inside it. But the inverse thinking is equally powerful and useful: take any arbitrary section of code you've written, and wrap a function declaration around it, which in effect "hides" the code.

The practical result is to create a scope bubble around the code in question, which means that any declarations (variable or function) in that code will now be tied to the scope of the new wrapping function, rather than the previously enclosing scope. In other words, you can "hide" variables and functions by enclosing them in the scope of a function.

Why would "hiding" variables and functions be a useful technique?

There's a variety of reasons motivating this scope-based hiding. They tend to arise from the software design principle "Principle of Least Privilege" [note-leastprivilege], also sometimes called "Least Authority" or "Least Exposure". This principle states that in the design of software, such as the API for a module/object, you should expose only what is minimally necessary, and "hide" everything else.

This principle extends to the choice of which scope to contain variables and functions. If all variables and functions were in the global scope, they would of course be accessible to any nested scope. But this would violate the "Least..." principle in that you are (likely) exposing many variables or functions which you should otherwise keep private, as proper use of the code would discourage access to those variables/functions.

For example:

```
function doSomething(a) {
    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

function doSomethingElse(a) {
    return a - 1;
}

var b;

doSomething( 2 ); // 15
```

In this snippet, the `b` variable and the `doSomethingElse(..)` function are likely "private" details of how `doSomething(..)` does its job. Giving the enclosing scope "access" to `b` and `doSomethingElse(..)` is not only unnecessary but also possibly "dangerous", in that they may be used in unexpected ways, intentionally or not, and this may violate pre-condition assumptions of `doSomething(..)`.

A more "proper" design would hide these private details inside the scope of `doSomething(..)`, such as:

```
function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }

    var b;

    b = a + doSomethingElse( a * 2 );

    console.log( (b * 3) );
}

doSomething( 2 ); // 15
```

Now, `b` and `doSomethingElse(..)` are not accessible to any outside influence, instead controlled only by `doSomething(..)`. The functionality and end-result has not been affected, but the design keeps private details private, which is usually considered better software.

Collision Avoidance

Another benefit of "hiding" variables and functions inside a scope is to avoid unintended collision between two different identifiers with the same name but different intended usages. Collision results often in unexpected overwriting of values.

For example:

```
function foo() {
    function bar(a) {
        i = 3; // changing the `i` in the enclosing scope's for-loop
        console.log( a + i );
    }

    for (var i=0; i<10; i++) {
        bar( i * 2 ); // oops, infinite loop ahead!
    }
}

foo();
```

The `i = 3` assignment inside of `bar(..)` overwrites, unexpectedly, the `i` that was declared in `foo(..)` at the for-loop. In this case, it will result in an infinite loop, because `i` is set to a fixed value of `3` and that will forever remain `< 10`.

The assignment inside `bar(..)` needs to declare a local variable to use, regardless of what identifier name is chosen. `var i = 3;` would fix the problem (and would create the previously mentioned "shadowed variable" declaration for `i`). An *additional*, not alternate, option is to pick another identifier name entirely, such as `var j = 3;`. But your software design may naturally call for the same identifier name, so utilizing scope to "hide" your inner declaration is your best/only option in that case.

Global "Namespaces"

A particularly strong example of (likely) variable collision occurs in the global scope. Multiple libraries loaded into your program can quite easily collide with each other if they don't properly hide their internal/private functions and variables.

Such libraries typically will create a single variable declaration, often an object, with a sufficiently unique name, in the global scope. This object is then used as a "namespace" for that library, where all specific exposures of functionality are made as properties off that object (namespace), rather than as top-level lexically scoped identifiers themselves.

For example:

```
var MyReallyCoolLibrary = {
    awesome: "stuff",
    doSomething: function() {
        // ...
    },
    doAnotherThing: function() {
        // ...
    }
};
```

Module Management

Another option for collision avoidance is the more modern "module" approach, using any of various dependency managers. Using these tools, no libraries ever add any identifiers to the global scope, but are instead required to have their identifier(s) be explicitly imported into another specific scope through usage of the dependency manager's various mechanisms.

It should be observed that these tools do not possess "magic" functionality that is exempt from lexical scoping rules. They simply use the rules of scoping as explained here to enforce that no identifiers are injected into any shared scope, and are instead kept in private, non-collision-susceptible scopes, which prevents any accidental scope collisions.

As such, you can code defensively and achieve the same results as the dependency managers do without actually needing to use them, if you so choose. See the Chapter 5 for more information about the module pattern.

Functions As Scopes

We've seen that we can take any snippet of code and wrap a function around it, and that effectively "hides" any enclosed variable or function declarations from the outside scope inside that function's inner scope.

For example:

```js
var a = 2;

function foo() { // <-- insert this

    var a = 3;
    console.log( a ); // 3

} // <-- and this
foo(); // <-- and this

console.log( a ); // 2
```

While this technique "works", it is not necessarily very ideal. There are a few problems it introduces. The first is that we have to declare a named-function `foo()`, which means that the identifier name `foo` itself "pollutes" the enclosing scope (global, in this case). We also have to explicitly call the function by name (`foo()`) so that the wrapped code actually executes.

It would be more ideal if the function didn't need a name (or, rather, the name didn't pollute the enclosing scope), and if the function could automatically be executed.

Fortunately, JavaScript offers a solution to both problems.

```js
var a = 2;

(function foo(){ // <-- insert this

    var a = 3;
    console.log( a ); // 3

})(); // <-- and this

console.log( a ); // 2
```

Let's break down what's happening here.

First, notice that the wrapping function statement starts with `(function...` as opposed to just `function...`. While this may seem like a minor detail, it's actually a major change. Instead of treating the function as a standard declaration, the function is treated as a function-expression.

Note: The easiest way to distinguish declaration vs. expression is the position of the word "function" in the statement (not just a line, but a distinct statement). If "function" is the very first thing in the statement, then it's a function declaration. Otherwise, it's a function expression.

The key difference we can observe here between a function declaration and a function expression relates to where its name is bound as an identifier.

Compare the previous two snippets. In the first snippet, the name `foo` is bound in the enclosing scope, and we call it directly with `foo()`. In the second snippet, the name `foo` is not bound in the enclosing scope, but instead is bound only inside of its own function.

In other words, `(function foo(){ .. })` as an expression means the identifier `foo` is found *only* in the scope where the `..` indicates, not in the outer scope. Hiding the name `foo` inside itself means it does not pollute the enclosing scope unnecessarily.

Anonymous vs. Named

You are probably most familiar with function expressions as callback parameters, such as:

```
setTimeout( function(){
    console.log("I waited 1 second!");
}, 1000 );
```

This is called an "anonymous function expression", because `function()...` has no name identifier on it. Function expressions can be anonymous, but function declarations cannot omit the name -- that would be illegal JS grammar.

Anonymous function expressions are quick and easy to type, and many libraries and tools tend to encourage this idiomatic style of code. However, they have several draw-backs to consider:

1. Anonymous functions have no useful name to display in stack traces, which can make debugging more difficult.
2. Without a name, if the function needs to refer to itself, for recursion, etc., the **deprecated** `arguments.callee` reference is unfortunately required. Another example of needing to self-reference is when an event handler function wants to unbind itself after it fires.
3. Anonymous functions omit a name that is often helpful in providing more readable/understandable code. A descriptive name helps self-document the code in question.

Inline function expressions are powerful and useful -- the question of anonymous vs. named doesn't detract from that. Providing a name for your function expression quite effectively addresses all these draw-backs, but has no tangible downsides. The best practice is to always name your function expressions:

```
setTimeout( function timeoutHandler(){ // <-- Look, I have a name!
    console.log( "I waited 1 second!" );
}, 1000 );
```

Invoking Function Expressions Immediately

```
var a = 2;

(function foo(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

Now that we have a function as an expression by virtue of wrapping it in a `( )` pair, we can execute that function by adding another `()` on the end, like `(function foo(){ .. })()`. The first enclosing `( )` pair makes the function an expression, and the second `()` executes the function.

This pattern is so common, a few years ago the community agreed on a term for it: **IIFE**, which stands for **I**mmediately **I**nvoked **F**unction **E**xpression.

Of course, IIFE's don't need names, necessarily -- the most common form of IIFE is to use an anonymous function expression. While certainly less common, naming an IIFE has all the aforementioned benefits over anonymous function expressions, so it's a good practice to adopt.

```
var a = 2;

(function IIFE(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

There's a slight variation on the traditional IIFE form, which some prefer: `(function(){ .. }())`. Look closely to see the difference. In the first form, the function expression is wrapped in `( )`, and then the invoking `()` pair is on the outside right after it. In the second form,

the invoking `()` pair is moved to the inside of the outer `( )` wrapping pair.

These two forms are identical in functionality. **It's purely a stylistic choice which you prefer.**

Another variation on IIFE's which is quite common is to use the fact that they are, in fact, just function calls, and pass in argument(s).

For instance:

```
var a = 2;

(function IIFE( global ){

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

})( window );

console.log( a ); // 2
```

We pass in the `window` object reference, but we name the parameter `global`, so that we have a clear stylistic delineation for global vs. non-global references. Of course, you can pass in anything from an enclosing scope you want, and you can name the parameter(s) anything that suits you. This is mostly just stylistic choice.

Another application of this pattern addresses the (minor niche) concern that the default `undefined` identifier might have its value incorrectly overwritten, causing unexpected results. By naming a parameter `undefined`, but not passing any value for that argument, we can guarantee that the `undefined` identifier is in fact the undefined value in a block of code:

```
undefined = true; // setting a land-mine for other code! avoid!

(function IIFE( undefined ){

    var a;
    if (a === undefined) {
        console.log( "Undefined is safe here!" );
    }

})();
```

Still another variation of the IIFE inverts the order of things, where the function to execute is given second, *after* the invocation and parameters to pass to it. This pattern is used in the UMD (Universal Module Definition) project. Some people find it a little cleaner to understand,

though it is slightly more verbose.

```
var a = 2;

(function IIFE( def ){
    def( window );
})(function def( global ){

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

});
```

The `def` function expression is defined in the second-half of the snippet, and then passed as a parameter (also called `def`) to the `IIFE` function defined in the first half of the snippet. Finally, the parameter `def` (the function) is invoked, passing `window` in as the `global` parameter.

Blocks As Scopes

While functions are the most common unit of scope, and certainly the most wide-spread of the design approaches in the majority of JS in circulation, other units of scope are possible, and the usage of these other scope units can lead to even better, cleaner to maintain code.

Many languages other than JavaScript support Block Scope, and so developers from those languages are accustomed to the mindset, whereas those who've primarily only worked in JavaScript may find the concept slightly foreign.

But even if you've never written a single line of code in block-scoped fashion, you are still probably familiar with this extremely common idiom in JavaScript:

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

We declare the variable `i` directly inside the for-loop head, most likely because our *intent* is to use `i` only within the context of that for-loop, and essentially ignore the fact that the variable actually scopes itself to the enclosing scope (function or global).

That's what block-scoping is all about. Declaring variables as close as possible, as local as possible, to where they will be used. Another example:

```
var foo = true;

if (foo) {
    var bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}
```

We are using a `bar` variable only in the context of the if-statement, so it makes a kind of sense that we would declare it inside the if-block. However, where we declare variables is not relevant when using `var`, because they will always belong to the enclosing scope. This snippet is essentially "fake" block-scoping, for stylistic reasons, and relying on self-enforcement not to accidentally use `bar` in another place in that scope.

Block scope is a tool to extend the earlier "Principle of Least ~~Privilege~~ Exposure" note-leastprivilege from hiding information in functions to hiding information in blocks of our code.

Consider the for-loop example again:

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

Why pollute the entire scope of a function with the `i` variable that is only going to be (or only *should be*, at least) used for the for-loop?

But more importantly, developers may prefer to *check* themselves against accidentally (re)using variables outside of their intended purpose, such as being issued an error about an unknown variable if you try to use it in the wrong place. Block-scoping (if it were possible) for the `i` variable would make `i` available only for the for-loop, causing an error if `i` is accessed elsewhere in the function. This helps ensure variables are not re-used in confusing or hard-to-maintain ways.

But, the sad reality is that, on the surface, JavaScript has no facility for block scope.

That is, until you dig a little further.

`with`

We learned about `with` in Chapter 2. While it is a frowned upon construct, it *is* an example of (a form of) block scope, in that the scope that is created from the object only exists for the lifetime of that `with` statement, and not in the enclosing scope.

`try/catch`

It's a *very* little known fact that JavaScript in ES3 specified the variable declaration in the `catch` clause of a `try/catch` to be block-scoped to the `catch` block.

For instance:

```
try {
    undefined(); // illegal operation to force an exception!
}
catch (err) {
    console.log( err ); // works!
}

console.log( err ); // ReferenceError: `err` not found
```

As you can see, `err` exists only in the `catch` clause, and throws an error when you try to reference it elsewhere.

Note: While this behavior has been specified and true of practically all standard JS environments (except perhaps old IE), many linters seem to still complain if you have two or more `catch` clauses in the same scope which each declare their error variable with the same identifier name. This is not actually a re-definition, since the variables are safely block-scoped, but the linters still seem to, annoyingly, complain about this fact.

To avoid these unnecessary warnings, some devs will name their `catch` variables `err1`, `err2`, etc. Other devs will simply turn off the linting check for duplicate variable names.

The block-scoping nature of `catch` may seem like a useless academic fact, but see Appendix B for more information on just how useful it might be.

`let`

Thus far, we've seen that JavaScript only has some strange niche behaviors which expose block scope functionality. If that were all we had, and *it was* for many, many years, then block scoping would not be terribly useful to the JavaScript developer.

Fortunately, ES6 changes that, and introduces a new keyword `let` which sits alongside `var` as another way to declare variables.

The `let` keyword attaches the variable declaration to the scope of whatever block (commonly a `{ .. }` pair) it's contained in. In other words, `let` implicitly hijacks any block's scope for its variable declaration.

```
var foo = true;

if (foo) {
    let bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}

console.log( bar ); // ReferenceError
```

Using `let` to attach a variable to an existing block is somewhat implicit. It can confuse if you're not paying close attention to which blocks have variables scoped to them, and are in the habit of moving blocks around, wrapping them in other blocks, etc., as you develop and evolve code.

Creating explicit blocks for block-scoping can address some of these concerns, making it more obvious where variables are attached and not. Usually, explicit code is preferable over implicit or subtle code. This explicit block-scoping style is easy to achieve, and fits more naturally with how block-scoping works in other languages:

```
var foo = true;

if (foo) {
    { // <-- explicit block
        let bar = foo * 2;
        bar = something( bar );
        console.log( bar );
    }
}

console.log( bar ); // ReferenceError
```

We can create an arbitrary block for `let` to bind to by simply including a `{ .. }` pair anywhere a statement is valid grammar. In this case, we've made an explicit block *inside* the if-statement, which may be easier as a whole block to move around later in refactoring, without affecting the position and semantics of the enclosing if-statement.

Note: For another way to express explicit block scopes, see Appendix B.

In Chapter 4, we will address hoisting, which talks about declarations being taken as existing for the entire scope in which they occur.

However, declarations made with `let` will *not* hoist to the entire scope of the block they appear in. Such declarations will not observably "exist" in the block until the declaration statement.

```
{
   console.log( bar ); // ReferenceError!
   let bar = 2;
}
```

Garbage Collection

Another reason block-scoping is useful relates to closures and garbage collection to reclaim memory. We'll briefly illustrate here, but the closure mechanism is explained in detail in Chapter 5.

Consider:

```
function process(data) {
    // do something interesting
}

var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

The `click` function click handler callback doesn't *need* the `someReallyBigData` variable at all. That means, theoretically, after `process(..)` runs, the big memory-heavy data structure could be garbage collected. However, it's quite likely (though implementation dependent) that the JS engine will still have to keep the structure around, since the `click` function has a closure over the entire scope.

Block-scoping can address this concern, making it clearer to the engine that it does not need to keep `someReallyBigData` around:

```
function process(data) {
    // do something interesting
}

// anything declared inside this block can go away after!
{
    let someReallyBigData = { .. };

    process( someReallyBigData );
}

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

Declaring explicit blocks for variables to locally bind to is a powerful tool that you can add to your code toolbox.

`let` Loops

A particular case where `let` shines is in the for-loop case as we discussed previously.

```
for (let i=0; i<10; i++) {
    console.log( i );
}

console.log( i ); // ReferenceError
```

Not only does `let` in the for-loop header bind the `i` to the for-loop body, but in fact, it **re-binds it** to each *iteration* of the loop, making sure to re-assign it the value from the end of the previous loop iteration.

Here's another way of illustrating the per-iteration binding behavior that occurs:

```
{
    let j;
    for (j=0; j<10; j++) {
        let i = j; // re-bound for each iteration!
        console.log( i );
    }
}
```

The reason why this per-iteration binding is interesting will become clear in Chapter 5 when we discuss closures.

Because `let` declarations attach to arbitrary blocks rather than to the enclosing function's scope (or global), there can be gotchas where existing code has a hidden reliance on function-scoped `var` declarations, and replacing the `var` with `let` may require additional care when refactoring code.

Consider:

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    if (baz > bar) {
        console.log( baz );
    }

    // ...
}
```

This code is fairly easily re-factored as:

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    // ...
}

if (baz > bar) {
    console.log( baz );
}
```

But, be careful of such changes when using block-scoped variables:

```
var foo = true, baz = 10;

if (foo) {
    let bar = 3;

    if (baz > bar) { // <-- don't forget `bar` when moving!
        console.log( baz );
    }
}
```

See Appendix B for an alternate (more explicit) style of block-scoping which may provide easier to maintain/refactor code that's more robust to these scenarios.

`const`

In addition to `let`, ES6 introduces `const`, which also creates a block-scoped variable, but whose value is fixed (constant). Any attempt to change that value at a later time results in an error.

```
var foo = true;

if (foo) {
    var a = 2;
    const b = 3; // block-scoped to the containing `if`

    a = 3; // just fine!
    b = 4; // error!
}

console.log( a ); // 3
console.log( b ); // ReferenceError!
```

Review (TL;DR)

Functions are the most common unit of scope in JavaScript. Variables and functions that are declared inside another function are essentially "hidden" from any of the enclosing "scopes", which is an intentional design principle of good software.

But functions are by no means the only unit of scope. Block-scope refers to the idea that variables and functions can belong to an arbitrary block (generally, any `{ .. }` pair) of code, rather than only to the enclosing function.

Starting with ES3, the `try/catch` structure has block-scope in the `catch` clause.

In ES6, the `let` keyword (a cousin to the `var` keyword) is introduced to allow declarations of variables in any arbitrary block of code. `if (..) { let a = 2; }` will declare a variable `a` that essentially hijacks the scope of the `if` 's `{ .. }` block and attaches itself there.

Though some seem to believe so, block scope should not be taken as an outright replacement of `var` function scope. Both functionalities co-exist, and developers can and should use both function-scope and block-scope techniques where respectively appropriate to produce better, more readable/maintainable code.

> note-leastprivilege. Principle of Least Privilege ↩

You Don't Know JS: Scope & Closures

Chapter 4: Hoisting

By now, you should be fairly comfortable with the idea of scope, and how variables are attached to different levels of scope depending on where and how they are declared. Both function scope and block scope behave by the same rules in this regard: any variable declared within a scope is attached to that scope.

But there's a subtle detail of how scope attachment works with declarations that appear in various locations within a scope, and that detail is what we will examine here.

Chicken Or The Egg?

There's a temptation to think that all of the code you see in a JavaScript program is interpreted line-by-line, top-down in order, as the program executes. While that is substantially true, there's one part of that assumption which can lead to incorrect thinking about your program.

Consider this code:

```
a = 2;

var a;

console.log( a );
```

What do you expect to be printed in the `console.log(..)` statement?

Many developers would expect `undefined`, since the `var a` statement comes after the `a = 2`, and it would seem natural to assume that the variable is re-defined, and thus assigned the default `undefined`. However, the output will be `2`.

Consider another piece of code:

```
console.log( a );

var a = 2;
```

You might be tempted to assume that, since the previous snippet exhibited some less-than-top-down looking behavior, perhaps in this snippet, `2` will also be printed. Others may think that since the `a` variable is used before it is declared, this must result in a `ReferenceError` being thrown.

Unfortunately, both guesses are incorrect. `undefined` is the output.

So, what's going on here? It would appear we have a chicken-and-the-egg question. Which comes first, the declaration ("egg"), or the assignment ("chicken")?

The Compiler Strikes Again

To answer this question, we need to refer back to Chapter 1, and our discussion of compilers. Recall that the *Engine* actually will compile your JavaScript code before it interprets it. Part of the compilation phase was to find and associate all declarations with their appropriate scopes. Chapter 2 showed us that this is the heart of Lexical Scope.

So, the best way to think about things is that all declarations, both variables and functions, are processed first, before any part of your code is executed.

When you see `var a = 2;`, you probably think of that as one statement. But JavaScript actually thinks of it as two statements: `var a;` and `a = 2;`. The first statement, the declaration, is processed during the compilation phase. The second statement, the assignment, is left **in place** for the execution phase.

Our first snippet then should be thought of as being handled like this:

```
var a;
```

```
a = 2;

console.log( a );
```

...where the first part is the compilation and the second part is the execution.

Similarly, our second snippet is actually processed as:

```
var a;
```

```
console.log( a );

a = 2;
```

So, one way of thinking, sort of metaphorically, about this process, is that variable and function declarations are "moved" from where they appear in the flow of the code to the top of the code. This gives rise to the name "Hoisting".

In other words, **the egg (declaration) comes before the chicken (assignment)**.

Note: Only the declarations themselves are hoisted, while any assignments or other executable logic are left *in place*. If hoisting were to re-arrange the executable logic of our code, that could wreak havoc.

```
foo();

function foo() {
    console.log( a ); // undefined

    var a = 2;
}
```

The function `foo`'s declaration (which in this case *includes* the implied value of it as an actual function) is hoisted, such that the call on the first line is able to execute.

It's also important to note that hoisting is **per-scope**. So while our previous snippets were simplified in that they only included global scope, the `foo(..)` function we are now examining itself exhibits that `var a` is hoisted to the top of `foo(..)` (not, obviously, to the top of the program). So the program can perhaps be more accurately interpreted like this:

```
function foo() {
    var a;

    console.log( a ); // undefined

    a = 2;
}

foo();
```

Function declarations are hoisted, as we just saw. But function expressions are not.

```
foo(); // not ReferenceError, but TypeError!

var foo = function bar() {
    // ...
};
```

The variable identifier `foo` is hoisted and attached to the enclosing scope (global) of this program, so `foo()` doesn't fail as a `ReferenceError`. But `foo` has no value yet (as it would if it had been a true function declaration instead of expression). So, `foo()` is attempting to invoke the `undefined` value, which is a `TypeError` illegal operation.

Also recall that even though it's a named function expression, the name identifier is not available in the enclosing scope:

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {
    // ...
};
```

This snippet is more accurately interpreted (with hoisting) as:

```
var foo;

foo(); // TypeError
bar(); // ReferenceError

foo = function() {
    var bar = ...self...
    // ...
}
```

Functions First

Both function declarations and variable declarations are hoisted. But a subtle detail (that *can* show up in code with multiple "duplicate" declarations) is that functions are hoisted first, and then variables.

Consider:

```js
foo(); // 1

var foo;

function foo() {
    console.log( 1 );
}

foo = function() {
    console.log( 2 );
};
```

`1` is printed instead of `2`! This snippet is interpreted by the *Engine* as:

```js
function foo() {
    console.log( 1 );
}

foo(); // 1

foo = function() {
    console.log( 2 );
};
```

Notice that `var foo` was the duplicate (and thus ignored) declaration, even though it came before the `function foo()...` declaration, because function declarations are hoisted before normal variables.

While multiple/duplicate `var` declarations are effectively ignored, subsequent function declarations *do* override previous ones.

```js
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

While this all may sound like nothing more than interesting academic trivia, it highlights the fact that duplicate definitions in the same scope are a really bad idea and will often lead to confusing results.

Function declarations that appear inside of normal blocks typically hoist to the enclosing scope, rather than being conditional as this code implies:

```
foo(); // "b"

var a = true;
if (a) {
   function foo() { console.log( "a" ); }
}
else {
   function foo() { console.log( "b" ); }
}
```

However, it's important to note that this behavior is not reliable and is subject to change in future versions of JavaScript, so it's probably best to avoid declaring functions in blocks.

Review (TL;DR)

We can be tempted to look at `var a = 2;` as one statement, but the JavaScript *Engine* does not see it that way. It sees `var a` and `a = 2` as two separate statements, the first one a compiler-phase task, and the second one an execution-phase task.

What this leads to is that all declarations in a scope, regardless of where they appear, are processed *first* before the code itself is executed. You can visualize this as declarations (variables and functions) being "moved" to the top of their respective scopes, which we call "hoisting".

Declarations themselves are hoisted, but assignments, even assignments of function expressions, are *not* hoisted.

Be careful about duplicate declarations, especially mixed between normal var declarations and function declarations -- peril awaits if you do!

You Don't Know JS: Scope & Closures

Chapter 5: Scope Closure

We arrive at this point with hopefully a very healthy, solid understanding of how scope works.

We turn our attention to an incredibly important, but persistently elusive, *almost mythological*, part of the language: **closure**. If you have followed our discussion of lexical scope thus far, the payoff is that closure is going to be, largely, anti-climatic, almost self-obvious. *There's a man behind the wizard's curtain, and we're about to see him*. No, his name is not Crockford!

If however you have nagging questions about lexical scope, now would be a good time to go back and review Chapter 2 before proceeding.

Enlightenment

For those who are somewhat experienced in JavaScript, but have perhaps never fully grasped the concept of closures, *understanding closure* can seem like a special nirvana that one must strive and sacrifice to attain.

I recall years back when I had a firm grasp on JavaScript, but had no idea what closure was. The hint that there was *this other side* to the language, one which promised even more capability than I already possessed, teased and taunted me. I remember reading through the source code of early frameworks trying to understand how it actually worked. I remember the first time something of the "module pattern" began to emerge in my mind. I remember the *a-ha!* moments quite vividly.

What I didn't know back then, what took me years to understand, and what I hope to impart to you presently, is this secret: **closure is all around you in JavaScript, you just have to recognize and embrace it.** Closures are not a special opt-in tool that you must learn new syntax and patterns for. No, closures are not even a weapon that you must learn to wield and master as Luke trained in The Force.

Closures happen as a result of writing code that relies on lexical scope. They just happen. You do not even really have to intentionally create closures to take advantage of them. Closures are created and used for you all over your code. What you are *missing* is the proper mental context to recognize, embrace, and leverage closures for your own will.

The enlightenment moment should be: **oh, closures are already occurring all over my code, I can finally *see* them now.** Understanding closures is like when Neo sees the Matrix for the first time.

Nitty Gritty

OK, enough hyperbole and shameless movie references.

Here's a down-n-dirty definition of what you need to know to understand and recognize closures:

> Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.

Let's jump into some code to illustrate that definition.

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a ); // 2
    }

    bar();
}

foo();
```

This code should look familiar from our discussions of Nested Scope. Function `bar()` has *access* to the variable `a` in the outer enclosing scope because of lexical scope look-up rules (in this case, it's an RHS reference look-up).

Is this "closure"?

Well, technically... *perhaps*. But by our what-you-need-to-know definition above... *not exactly*. I think the most accurate way to explain `bar()` referencing `a` is via lexical scope look-up rules, and those rules are *only* (an important!) **part** of what closure is.

From a purely academic perspective, what is said of the above snippet is that the function `bar()` has a *closure* over the scope of `foo()` (and indeed, even over the rest of the scopes it has access to, such as the global scope in our case). Put slightly differently, it's said that `bar()` closes over the scope of `foo()`. Why? Because `bar()` appears nested inside of `foo()`. Plain and simple.

But, closure defined in this way is not directly *observable*, nor do we see closure *exercised* in that snippet. We clearly see lexical scope, but closure remains sort of a mysterious shifting shadow behind the code.

Let us then consider code which brings closure into full light:

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a );
    }

    return bar;
}

var baz = foo();

baz(); // 2 -- Whoa, closure was just observed, man.
```

The function `bar()` has lexical scope access to the inner scope of `foo()`. But then, we take `bar()`, the function itself, and pass it *as* a value. In this case, we `return` the function object itself that `bar` references.

After we execute `foo()`, we assign the value it returned (our inner `bar()` function) to a variable called `baz`, and then we actually invoke `baz()`, which of course is invoking our inner function `bar()`, just by a different identifier reference.

`bar()` is executed, for sure. But in this case, it's executed *outside* of its declared lexical scope.

After `foo()` executed, normally we would expect that the entirety of the inner scope of `foo()` would go away, because we know that the *Engine* employs a *Garbage Collector* that comes along and frees up memory once it's no longer in use. Since it would appear that the contents of `foo()` are no longer in use, it would seem natural that they should be considered *gone*.

But the "magic" of closures does not let this happen. That inner scope is in fact *still* "in use", and thus does not go away. Who's using it? **The function `bar()` itself**.

By virtue of where it was declared, `bar()` has a lexical scope closure over that inner scope of `foo()`, which keeps that scope alive for `bar()` to reference at any later time.

`bar()` still has a reference to that scope, and that reference is called closure.

So, a few microseconds later, when the variable `baz` is invoked (invoking the inner function we initially labeled `bar`), it duly has *access* to author-time lexical scope, so it can access the variable `a` just as we'd expect.

The function is being invoked well outside of its author-time lexical scope. **Closure** lets the function continue to access the lexical scope it was defined in at author-time.

Of course, any of the various ways that functions can be *passed around* as values, and indeed invoked in other locations, are all examples of observing/exercising closure.

```
function foo() {
    var a = 2;

    function baz() {
        console.log( a ); // 2
    }

    bar( baz );
}

function bar(fn) {
    fn(); // look ma, I saw closure!
}
```

We pass the inner function `baz` over to `bar`, and call that inner function (labeled `fn` now), and when we do, its closure over the inner scope of `foo()` is observed, by accessing `a`.

These passings-around of functions can be indirect, too.

```
var fn;

function foo() {
    var a = 2;

    function baz() {
        console.log( a );
    }

    fn = baz; // assign `baz` to global variable
}

function bar() {
    fn(); // look ma, I saw closure!
}

foo();

bar(); // 2
```

Whatever facility we use to *transport* an inner function outside of its lexical scope, it will maintain a scope reference to where it was originally declared, and wherever we execute it, that closure will be exercised.

Now I Can See

The previous code snippets are somewhat academic and artificially constructed to illustrate *using closure*. But I promised you something more than just a cool new toy. I promised that closure was something all around you in your existing code. Let us now *see* that truth.

```
function wait(message) {

    setTimeout( function timer(){
        console.log( message );
    }, 1000 );

}

wait( "Hello, closure!" );
```

We take an inner function (named `timer`) and pass it to `setTimeout(..)` . But `timer` has a scope closure over the scope of `wait(..)` , indeed keeping and using a reference to the variable `message` .

A thousand milliseconds after we have executed `wait(..)`, and its inner scope should otherwise be long gone, that inner function `timer` still has closure over that scope.

Deep down in the guts of the *Engine*, the built-in utility `setTimeout(..)` has reference to some parameter, probably called `fn` or `func` or something like that. *Engine* goes to invoke that function, which is invoking our inner `timer` function, and the lexical scope reference is still intact.

Closure.

Or, if you're of the jQuery persuasion (or any JS framework, for that matter):

```
function setupBot(name,selector) {
    $( selector ).click( function activator(){
        console.log( "Activating: " + name );
    } );
}

setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

I am not sure what kind of code you write, but I regularly write code which is responsible for controlling an entire global drone army of closure bots, so this is totally realistic!

(Some) joking aside, essentially *whenever* and *wherever* you treat functions (which access their own respective lexical scopes) as first-class values and pass them around, you are likely to see those functions exercising closure. Be that timers, event handlers, Ajax requests, cross-window messaging, web workers, or any of the other asynchronous (or synchronous!) tasks, when you pass in a *callback function*, get ready to sling some closure around!

Note: Chapter 3 introduced the IIFE pattern. While it is often said that IIFE (alone) is an example of observed closure, I would somewhat disagree, by our definition above.

```
var a = 2;

(function IIFE(){
    console.log( a );
})();
```

This code "works", but it's not strictly an observation of closure. Why? Because the function (which we named "IIFE" here) is not executed outside its lexical scope. It's still invoked right there in the same scope as it was declared (the enclosing/global scope that also holds `a`). `a` is found via normal lexical scope look-up, not really via closure.

While closure might technically be happening at declaration time, it is *not* strictly observable, and so, as they say, *it's a tree falling in the forest with no one around to hear it.*

Though an IIFE is not *itself* an example of closure, it absolutely creates scope, and it's one of the most common tools we use to create scope which can be closed over. So IIFEs are indeed heavily related to closure, even if not exercising closure themselves.

Put this book down right now, dear reader. I have a task for you. Go open up some of your recent JavaScript code. Look for your functions-as-values and identify where you are already using closure and maybe didn't even know it before.

I'll wait.

Now... you see!

Loops + Closure

The most common canonical example used to illustrate closure involves the humble for-loop.

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

Note: Linters often complain when you put functions inside of loops, because the mistakes of not understanding closure are **so common among developers**. We explain how to do so properly here, leveraging the full power of closure. But that subtlety is often lost on linters and they will complain regardless, assuming you don't *actually* know what you're doing.

The spirit of this code snippet is that we would normally *expect* for the behavior to be that the numbers "1", "2", .. "5" would be printed out, one at a time, one per second, respectively.

In fact, if you run this code, you get "6" printed out 5 times, at the one-second intervals.

Huh?

Firstly, let's explain where `6` comes from. The terminating condition of the loop is when `i` is *not* `<=5`. The first time that's the case is when `i` is 6. So, the output is reflecting the final value of the `i` after the loop terminates.

This actually seems obvious on second glance. The timeout function callbacks are all running well after the completion of the loop. In fact, as timers go, even if it was `setTimeout(.., 0)` on each iteration, all those function callbacks would still run strictly after the completion of the loop, and thus print `6` each time.

But there's a deeper question at play here. What's *missing* from our code to actually have it behave as we semantically have implied?

What's missing is that we are trying to *imply* that each iteration of the loop "captures" its own copy of `i`, at the time of the iteration. But, the way scope works, all 5 of those functions, though they are defined separately in each loop iteration, all **are closed over the same shared global scope**, which has, in fact, only one `i` in it.

Put that way, *of course* all functions share a reference to the same `i`. Something about the loop structure tends to confuse us into thinking there's something else more sophisticated at work. There is not. There's no difference than if each of the 5 timeout callbacks were just declared one right after the other, with no loop at all.

OK, so, back to our burning question. What's missing? We need more ~~cowbell~~ closured scope. Specifically, we need a new closured scope for each iteration of the loop.

We learned in Chapter 3 that the IIFE creates scope by declaring a function and immediately executing it.

Let's try:

```
for (var i=1; i<=5; i++) {
    (function(){
        setTimeout( function timer(){
            console.log( i );
        }, i*1000 );
    })();
}
```

Does that work? Try it. Again, I'll wait.

I'll end the suspense for you. **Nope.** But why? We now obviously have more lexical scope. Each timeout function callback is indeed closing over its own per-iteration scope created respectively by each IIFE.

It's not enough to have a scope to close over **if that scope is empty**. Look closely. Our IIFE is just an empty do-nothing scope. It needs *something* in it to be useful to us.

It needs its own variable, with a copy of the `i` value at each iteration.

```
for (var i=1; i<=5; i++) {
    (function(){
        var j = i;
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })();
}
```

Eureka! It works!

A slight variation some prefer is:

```
for (var i=1; i<=5; i++) {
    (function(j){
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })( i );
}
```

Of course, since these IIFEs are just functions, we can pass in `i`, and we can call it `j` if we prefer, or we can even call it `i` again. Either way, the code works now.

The use of an IIFE inside each iteration created a new scope for each iteration, which gave our timeout function callbacks the opportunity to close over a new scope for each iteration, one which had a variable with the right per-iteration value in it for us to access.

Problem solved!

Block Scoping Revisited

Look carefully at our analysis of the previous solution. We used an IIFE to create new scope per-iteration. In other words, we actually *needed* a per-iteration **block scope**. Chapter 3 showed us the `let` declaration, which hijacks a block and declares a variable right there in the block.

It essentially turns a block into a scope that we can close over. So, the following awesome code "just works":

```
for (var i=1; i<=5; i++) {
    let j = i; // yay, block-scope for closure!
    setTimeout( function timer(){
        console.log( j );
    }, j*1000 );
}
```

But, that's not all! (in my best Bob Barker voice). There's a special behavior defined for `let` declarations used in the head of a for-loop. This behavior says that the variable will be declared not just once for the loop, **but each iteration**. And, it will, helpfully, be initialized at each subsequent iteration with the value from the end of the previous iteration.

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

How cool is that? Block scoping and closure working hand-in-hand, solving all the world's problems. I don't know about you, but that makes me a happy JavaScripter.

Modules

There are other code patterns which leverage the power of closure but which do not on the surface appear to be about callbacks. Let's examine the most powerful of them: *the module*.

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

As this code stands right now, there's no observable closure going on. We simply have some private data variables `something` and `another`, and a couple of inner functions `doSomething()` and `doAnother()`, which both have lexical scope (and thus closure!) over the inner scope of `foo()`.

But now consider:

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

This is the pattern in JavaScript we call *module*. The most common way of implementing the module pattern is often called "Revealing Module", and it's the variation we present here.

Let's examine some things about this code.

Firstly, `CoolModule()` is just a function, but it *has to be invoked* for there to be a module instance created. Without the execution of the outer function, the creation of the inner scope and the closures would not occur.

Secondly, the `CoolModule()` function returns an object, denoted by the object-literal syntax `{ key: value, ... }`. The object we return has references on it to our inner functions, but *not* to our inner data variables. We keep those hidden and private. It's appropriate to think of this object return value as essentially a **public API for our module**.

This object return value is ultimately assigned to the outer variable `foo`, and then we can access those property methods on the API, like `foo.doSomething()`.

Note: It is not required that we return an actual object (literal) from our module. We could just return back an inner function directly. jQuery is actually a good example of this. The `jQuery` and `$` identifiers are the public API for the jQuery "module", but they are, themselves, just a function (which can itself have properties, since all functions are objects).

The `doSomething()` and `doAnother()` functions have closure over the inner scope of the module "instance" (arrived at by actually invoking `CoolModule()`). When we transport those functions outside of the lexical scope, by way of property references on the object we return, we have now set up a condition by which closure can be observed and exercised.

To state it more simply, there are two "requirements" for the module pattern to be exercised:

1. There must be an outer enclosing function, and it must be invoked at least once (each time creates a new module instance).
2. The enclosing function must return back at least one inner function, so that this inner function has closure over the private scope, and can access and/or modify that private state.

An object with a function property on it alone is not *really* a module. An object which is returned from a function invocation which only has data properties on it and no closured functions is not *really* a module, in the observable sense.

The code snippet above shows a standalone module creator called `CoolModule()` which can be invoked any number of times, each time creating a new module instance. A slight variation on this pattern is when you only care to have one instance, a "singleton" of sorts:

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Here, we turned our module function into an IIFE (see Chapter 3), and we *immediately* invoked it and assigned its return value directly to our single module instance identifier `foo` .

Modules are just functions, so they can receive parameters:

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }

    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );

foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

Another slight but powerful variation on the module pattern is to name the object you are returning as your public API:

```
var foo = (function CoolModule(id) {
    function change() {
        // modifying the public API
        publicAPI.identify = identify2;
    }

    function identify1() {
        console.log( id );
    }

    function identify2() {
        console.log( id.toUpperCase() );
    }

    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

By retaining an inner reference to the public API object inside your module instance, you can modify that module instance **from the inside**, including adding and removing methods, properties, *and* changing their values.

Modern Modules

Various module dependency loaders/managers essentially wrap up this pattern of module definition into a friendly API. Rather than examine any one particular library, let me present a *very simple* proof of concept **for illustration purposes (only)**:

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply( impl, deps );
    }

    function get(name) {
        return modules[name];
    }

    return {
        define: define,
        get: get
    };
})();
```

The key part of this code is `modules[name] = impl.apply(impl, deps)`. This is invoking the definition wrapper function for a module (passing in any dependencies), and storing the return value, the module's API, into an internal list of modules tracked by name.

And here's how I might use it to define some modules:

```
MyModules.define( "bar", [], function(){
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar){
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Both the "foo" and "bar" modules are defined with a function that returns a public API. "foo" even receives the instance of "bar" as a dependency parameter, and can use it accordingly.

Spend some time examining these code snippets to fully understand the power of closures put to use for our own good purposes. The key take-away is that there's not really any particular "magic" to module managers. They fulfill both characteristics of the module pattern I listed above: invoking a function definition wrapper, and keeping its return value as the API for that module.

In other words, modules are just modules, even if you put a friendly wrapper tool on top of them.

Future Modules

ES6 adds first-class syntax support for the concept of modules. When loaded via the module system, ES6 treats a file as a separate module. Each module can both import other modules or specific API members, as well export their own public API members.

Note: Function-based modules aren't a statically recognized pattern (something the compiler knows about), so their API semantics aren't considered until run-time. That is, you can actually modify a module's API during the run-time (see earlier `publicAPI` discussion).

By contrast, ES6 Module APIs are static (the APIs don't change at run-time). Since the compiler knows *that*, it can (and does!) check during (file loading and) compilation that a reference to a member of an imported module's API *actually exists*. If the API reference doesn't exist, the compiler throws an "early" error at compile-time, rather than waiting for traditional dynamic run-time resolution (and errors, if any).

ES6 modules **do not** have an "inline" format, they must be defined in separate files (one per module). The browsers/engines have a default "module loader" (which is overridable, but that's well-beyond our discussion here) which synchronously loads a module file when it's imported.

Consider:

bar.js

```
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;
```

foo.js

```
// import only `hello()` from the "bar" module
import hello from "bar";

var hungry = "hippo";

function awesome() {
    console.log(
        hello( hungry ).toUpperCase()
    );
}

export awesome;
```

```
// import the entire "foo" and "bar" modules
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Note: Separate files **"foo.js"** and **"bar.js"** would need to be created, with the contents as shown in the first two snippets, respectively. Then, your program would load/import those modules to use them, as shown in the third snippet.

`import` imports one or more members from a module's API into the current scope, each to a bound variable (`hello` in our case). `module` imports an entire module API to a bound variable (`foo` , `bar` in our case). `export` exports an identifier (variable, function) to the public API for the current module. These operators can be used as many times in a module's definition as is necessary.

The contents inside the *module file* are treated as if enclosed in a scope closure, just like with the function-closure modules seen earlier.

Review (TL;DR)

Closure seems to the un-enlightened like a mystical world set apart inside of JavaScript which only the few bravest souls can reach. But it's actually just a standard and almost obvious fact of how we write code in a lexically scoped environment, where functions are values and can be passed around at will.

Closure is when a function can remember and access its lexical scope even when it's invoked outside its lexical scope.

Closures can trip us up, for instance with loops, if we're not careful to recognize them and how they work. But they are also an immensely powerful tool, enabling patterns like *modules* in their various forms.

Modules require two key characteristics: 1) an outer wrapping function being invoked, to create the enclosing scope 2) the return value of the wrapping function must include reference to at least one inner function that then has closure over the private inner scope of the wrapper.

Now we can see closures all around our existing code, and we have the ability to recognize and leverage them to our own benefit!

You Don't Know JS: Scope & Closures

Appendix A: Dynamic Scope

In Chapter 2, we talked about "Dynamic Scope" as a contrast to the "Lexical Scope" model, which is how scope works in JavaScript (and in fact, most other languages).

We will briefly examine dynamic scope, to hammer home the contrast. But, more importantly, dynamic scope actually is a near cousin to another mechanism (`this`) in JavaScript, which we covered in the "*this & Object Prototypes*" title of this book series.

As we saw in Chapter 2, lexical scope is the set of rules about how the *Engine* can look-up a variable and where it will find it. The key characteristic of lexical scope is that it is defined at author-time, when the code is written (assuming you don't cheat with `eval()` or `with`).

Dynamic scope seems to imply, and for good reason, that there's a model whereby scope can be determined dynamically at runtime, rather than statically at author-time. That is in fact the case. Let's illustrate via code:

```
function foo() {
    console.log( a ); // 2
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

Lexical scope holds that the RHS reference to `a` in `foo()` will be resolved to the global variable `a` , which will result in value `2` being output.

Dynamic scope, by contrast, doesn't concern itself with how and where functions and scopes are declared, but rather **where they are called from**. In other words, the scope chain is based on the call-stack, not the nesting of scopes in code.

So, if JavaScript had dynamic scope, when `foo()` is executed, **theoretically** the code below would instead result in `3` as the output.

```
function foo() {
    console.log( a ); // 3  (not 2!)
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

How can this be? Because when `foo()` cannot resolve the variable reference for `a` , instead of stepping up the nested (lexical) scope chain, it walks up the call-stack, to find where `foo()` was *called from*. Since `foo()` was called from `bar()` , it checks the variables in scope for `bar()` , and finds an `a` there with value `3` .

Strange? You're probably thinking so, at the moment.

But that's just because you've probably only ever worked on (or at least deeply considered) code which is lexically scoped. So dynamic scoping seems foreign. If you had only ever written code in a dynamically scoped language, it would seem natural, and lexical scope would be the odd-ball.

To be clear, JavaScript **does not, in fact, have dynamic scope**. It has lexical scope. Plain and simple. But the `this` mechanism is kind of like dynamic scope.

The key contrast: **lexical scope is write-time, whereas dynamic scope (and `this` !) are runtime**. Lexical scope cares *where a function was declared*, but dynamic scope cares where a function was *called from*.

Finally: `this` cares *how a function was called*, which shows how closely related the `this` mechanism is to the idea of dynamic scoping. To dig more into `this` , read the title "*this & Object Prototypes*".

You Don't Know JS: Scope & Closures

Appendix B: Polyfilling Block Scope

In Chapter 3, we explored Block Scope. We saw that `with` and the `catch` clause are both tiny examples of block scope that have existed in JavaScript since at least the introduction of ES3.

But it's ES6's introduction of `let` that finally gives full, unfettered block-scoping capability to our code. There are many exciting things, both functionally and code-stylistically, that block scope will enable.

But what if we wanted to use block scope in pre-ES6 environments?

Consider this code:

```
{
    let a = 2;
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

This will work great in ES6 environments. But can we do so pre-ES6? `catch` is the answer.

```
try{throw 2}catch(a){
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

Whoa! That's some ugly, weird looking code. We see a `try/catch` that appears to forcibly throw an error, but the "error" it throws is just a value `2`, and then the variable declaration that receives it is in the `catch(a)` clause. Mind: blown.

That's right, the `catch` clause has block-scoping to it, which means it can be used as a polyfill for block scope in pre-ES6 environments.

"But...", you say. "...no one wants to write ugly code like that!" That's true. No one writes (some of) the code output by the CoffeeScript compiler, either. That's not the point.

The point is that tools can transpile ES6 code to work in pre-ES6 environments. You can write code using block-scoping, and benefit from such functionality, and let a build-step tool take care of producing code that will actually *work* when deployed.

This is actually the preferred migration path for all (ahem, most) of ES6: to use a code transpiler to take ES6 code and produce ES5-compatible code during the transition from pre-ES6 to ES6.

Traceur

Google maintains a project called "Traceur" note-traceur, which is exactly tasked with transpiling ES6 features into pre-ES6 (mostly ES5, but not all!) for general usage. The TC39 committee relies on this tool (and others) to test out the semantics of the features they specify.

What does Traceur produce from our snippet? You guessed it!

```
{
    try {
        throw undefined;
    } catch (a) {
        a = 2;
        console.log( a );
    }
}

console.log( a );
```

So, with the use of such tools, we can start taking advantage of block scope regardless of if we are targeting ES6 or not, because `try/catch` has been around (and worked this way) from ES3 days.

Implicit vs. Explicit Blocks

In Chapter 3, we identified some potential pitfalls to code maintainability/refactorability when we introduce block-scoping. Is there another way to take advantage of block scope but to reduce this downside?

Consider this alternate form of `let`, called the "let block" or "let statement" (contrasted with "let declarations" from before).

```
let (a = 2) {
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

Instead of implicitly hijacking an existing block, the let-statement creates an explicit block for its scope binding. Not only does the explicit block stand out more, and perhaps fare more robustly in code refactoring, it produces somewhat cleaner code by, grammatically, forcing all the declarations to the top of the block. This makes it easier to look at any block and know what's scoped to it and not.

As a pattern, it mirrors the approach many people take in function-scoping when they manually move/hoist all their `var` declarations to the top of the function. The let-statement puts them there at the top of the block by intent, and if you don't use `let` declarations strewn throughout, your block-scoping declarations are somewhat easier to identify and maintain.

But, there's a problem. The let-statement form is not included in ES6. Neither does the official Traceur compiler accept that form of code.

We have two options. We can format using ES6-valid syntax and a little sprinkle of code discipline:

```
/*let*/ { let a = 2;
    console.log( a );
}

console.log( a ); // ReferenceError
```

But, tools are meant to solve our problems. So the other option is to write explicit let statement blocks, and let a tool convert them to valid, working code.

So, I built a tool called "let-er" [note-let_er] to address just this issue. *let-er* is a build-step code transpiler, but its only task is to find let-statement forms and transpile them. It will leave alone any of the rest of your code, including any let-declarations. You can safely use *let-er* as the first ES6 transpiler step, and then pass your code through something like Traceur if necessary.

Moreover, *let-er* has a configuration flag `--es6`, which when turned on (off by default), changes the kind of code produced. Instead of the `try/catch` ES3 polyfill hack, *let-er* would take our snippet and produce the fully ES6-compliant, non-hacky:

```
{
    let a = 2;
    console.log( a );
}

console.log( a ); // ReferenceError
```

So, you can start using *let-er* right away, and target all pre-ES6 environments, and when you only care about ES6, you can add the flag and instantly target only ES6.

And most importantly, **you can use the more preferable and more explicit let-statement form** even though it is not an official part of any ES version (yet).

Performance

Let me add one last quick note on the performance of `try/catch`, and/or to address the question, "why not just use an IIFE to create the scope?"

Firstly, the performance of `try/catch` *is* slower, but there's no reasonable assumption that it *has* to be that way, or even that it *always will be* that way. Since the official TC39-approved ES6 transpiler uses `try/catch`, the Traceur team has asked Chrome to improve the performance of `try/catch`, and they are obviously motivated to do so.

Secondly, IIFE is not a fair apples-to-apples comparison with `try/catch`, because a function wrapped around any arbitrary code changes the meaning, inside of that code, of `this`, `return`, `break`, and `continue`. IIFE is not a suitable general substitute. It could only be used manually in certain cases.

The question really becomes: do you want block-scoping, or not. If you do, these tools provide you that option. If not, keep using `var` and go on about your coding!

> note-traceur. Google Traceur ↩

> note-let_er. let-er ↩

You Don't Know JS: Scope & Closures

Appendix C: Lexical-this

Though this title does not address the `this` mechanism in any detail, there's one ES6 topic which relates `this` to lexical scope in an important way, which we will quickly examine.

ES6 adds a special syntactic form of function declaration called the "arrow function". It looks like this:

```
var foo = a => {
    console.log( a );
};

foo( 2 ); // 2
```

The so-called "fat arrow" is often mentioned as a short-hand for the *tediously verbose* (sarcasm) `function` keyword.

But there's something much more important going on with arrow-functions that has nothing to do with saving keystrokes in your declaration.

Briefly, this code suffers a problem:

```
var obj = {
    id: "awesome",
    cool: function coolFn() {
        console.log( this.id );
    }
};

var id = "not awesome";

obj.cool(); // awesome

setTimeout( obj.cool, 100 ); // not awesome
```

The problem is the loss of `this` binding on the `cool()` function. There are various ways to address that problem, but one often-repeated solution is `var self = this;`.

That might look like:

```
var obj = {
    count: 0,
    cool: function coolFn() {
        var self = this;

        if (self.count < 1) {
            setTimeout( function timer(){
                self.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

Without getting too much into the weeds here, the `var self = this` "solution" just dispenses with the whole problem of understanding and properly using `this` binding, and instead falls back to something we're perhaps more comfortable with: lexical scope. `self` becomes just an identifier that can be resolved via lexical scope and closure, and cares not what happened to the `this` binding along the way.

People don't like writing verbose stuff, especially when they do it over and over again. So, a motivation of ES6 is to help alleviate these scenarios, and indeed, *fix* common idiom problems, such as this one.

The ES6 solution, the arrow-function, introduces a behavior called "lexical this".

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( () => { // arrow-function ftw?
                this.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

The short explanation is that arrow-functions do not behave at all like normal functions when it comes to their `this` binding. They discard all the normal rules for `this` binding, and instead take on the `this` value of their immediate lexical enclosing scope, whatever it is.

So, in that snippet, the arrow-function doesn't get its `this` unbound in some unpredictable way, it just "inherits" the `this` binding of the `cool()` function (which is correct if we invoke it as shown!).

While this makes for shorter code, my perspective is that arrow-functions are really just codifying into the language syntax a common *mistake* of developers, which is to confuse and conflate "this binding" rules with "lexical scope" rules.

Put another way: why go to the trouble and verbosity of using the `this` style coding paradigm, only to cut it off at the knees by mixing it with lexical references. It seems natural to embrace one approach or the other for any given piece of code, and not mix them in the same piece of code.

Note: one other detraction from arrow-functions is that they are anonymous, not named. See Chapter 3 for the reasons why anonymous functions are less desirable than named functions.

A more appropriate approach, in my perspective, to this "problem", is to use and embrace the `this` mechanism correctly.

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( function timer(){
                this.count++; // `this` is safe because of `bind(..)`
                console.log( "more awesome" );
            }.bind( this ), 100 ); // look, `bind()`!
        }
    }
};

obj.cool(); // more awesome
```

Whether you prefer the new lexical-this behavior of arrow-functions, or you prefer the tried-and-true `bind()`, it's important to note that arrow-functions are **not** just about less typing of "function".

They have an *intentional behavioral difference* that we should learn and understand, and if we so choose, leverage.

Now that we fully understand lexical scoping (and closure!), understanding lexical-this should be a breeze!

You Don't Know JS: Scope & Closures

Appendix D: Acknowledgments

I have many people to thank for making this book title and the overall series happen.

First, I must thank my wife Christen Simpson, and my two kids Ethan and Emily, for putting up with Dad always pecking away at the computer. Even when not writing books, my obsession with JavaScript glues my eyes to the screen far more than it should. That time I borrow from my family is the reason these books can so deeply and completely explain JavaScript to you, the reader. I owe my family everything.

I'd like to thank my editors at O'Reilly, namely Simon St.Laurent and Brian MacDonald, as well as the rest of the editorial and marketing staff. They are fantastic to work with, and have been especially accommodating during this experiment into "open source" book writing, editing, and production.

Thank you to the many folks who have participated in making this book series better by providing editorial suggestions and corrections, including Shelley Powers, Tim Ferro, Evan Borden, Forrest L. Norvell, Jennifer Davis, Jesse Harlin, and many others. A big thank you to Shane Hudson for writing the Foreword for this title.

Thank you to the countless folks in the community, including members of the TC39 committee, who have shared so much knowledge with the rest of us, and especially tolerated my incessant questions and explorations with patience and detail. John-David Dalton, Juriy "kangax" Zaytsev, Mathias Bynens, Axel Rauschmayer, Nicholas Zakas, Angus Croll, Reginald Braithwaite, Dave Herman, Brendan Eich, Allen Wirfs-Brock, Bradley Meck, Domenic Denicola, David Walsh, Tim Disney, Peter van der Zee, Andrea Giammarchi, Kit Cambridge, Eric Elliott, and so many others, I can't even scratch the surface.

The *You Don't Know JS* book series was born on Kickstarter, so I also wish to thank all my (nearly) 500 generous backers, without whom this book series could not have happened:

> Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt

> Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsden, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel בר-לבב Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuț Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie,

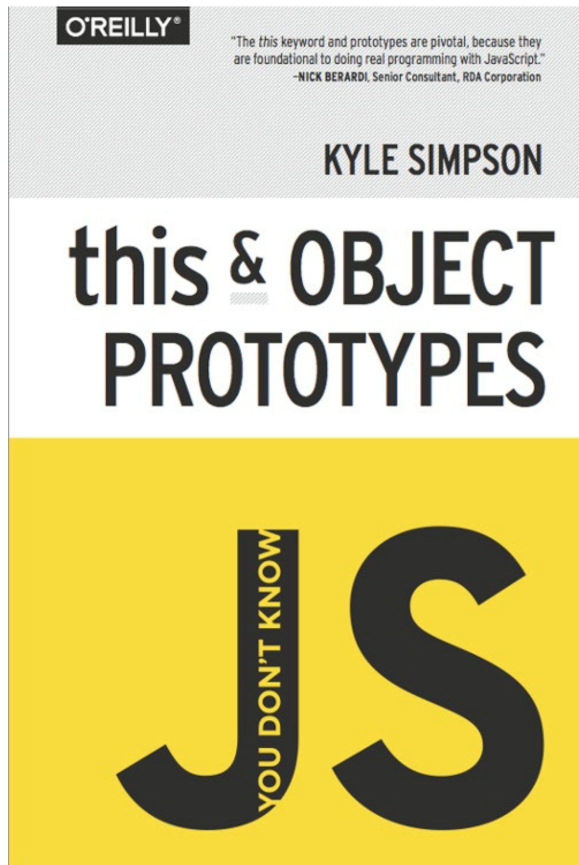
> Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave ♥♫★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry,

> Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard

This book series is being produced in an open source fashion, including editing and production. We owe GitHub a debt of gratitude for making that sort of thing possible for the community!

Thank you again to all the countless folks I didn't name but who I nonetheless owe thanks. May this book series be "owned" by all of us and serve to contribute to increasing awareness and understanding of the JavaScript language, to the benefit of all current and future community contributors.

You Don't Know JS: *this* & Object Prototypes



Purchase digital/print copy from O'Reilly

Table of Contents

You Don't Know JS: *this* & Object Prototypes

Chapter 1: `this` Or That?

One of the most confused mechanisms in JavaScript is the `this` keyword. It's a special identifier keyword that's automatically defined in the scope of every function, but what exactly it refers to bedevils even seasoned JavaScript developers.

> Any sufficiently *advanced* technology is indistinguishable from magic. -- Arthur C. Clarke

JavaScript's `this` mechanism isn't actually *that* advanced, but developers often paraphrase that quote in their own mind by inserting "complex" or "confusing", and there's no question that without lack of clear understanding, `this` can seem downright magical in *your* confusion.

Note: The word "this" is a terribly common pronoun in general discourse. So, it can be very difficult, especially verbally, to determine whether we are using "this" as a pronoun or using it to refer to the actual keyword identifier. For clarity, I will always use `this` to refer to the special keyword, and "this" or *this* or this otherwise.

Why `this`?

If the `this` mechanism is so confusing, even to seasoned JavaScript developers, one may wonder why it's even useful? Is it more trouble than it's worth? Before we jump into the *how*, we should examine the *why*.

Let's try to illustrate the motivation and utility of `this`:

```
function identify() {
    return this.name.toUpperCase();
}

function speak() {
    var greeting = "Hello, I'm " + identify.call( this );
    console.log( greeting );
}

var me = {
    name: "Kyle"
};

var you = {
    name: "Reader"
};

identify.call( me ); // KYLE
identify.call( you ); // READER

speak.call( me ); // Hello, I'm KYLE
speak.call( you ); // Hello, I'm READER
```

If the *how* of this snippet confuses you, don't worry! We'll get to that shortly. Just set those questions aside briefly so we can look into the *why* more clearly.

This code snippet allows the `identify()` and `speak()` functions to be re-used against multiple *context* (`me` and `you`) objects, rather than needing a separate version of the function for each object.

Instead of relying on `this`, you could have explicitly passed in a context object to both `identify()` and `speak()`.

```
function identify(context) {
    return context.name.toUpperCase();
}

function speak(context) {
    var greeting = "Hello, I'm " + identify( context );
    console.log( greeting );
}

identify( you ); // READER
speak( me ); // Hello, I'm KYLE
```

However, the `this` mechanism provides a more elegant way of implicitly "passing along" an object reference, leading to cleaner API design and easier re-use.

The more complex your usage pattern is, the more clearly you'll see that passing context around as an explicit parameter is often messier than passing around a `this` context. When we explore objects and prototypes, you will see the helpfulness of a collection of functions being able to automatically reference the proper context object.

Confusions

We'll soon begin to explain how `this` *actually* works, but first we must dispel some misconceptions about how it *doesn't* actually work.

The name "this" creates confusion when developers try to think about it too literally. There are two meanings often assumed, but both are incorrect.

Itself

The first common temptation is to assume `this` refers to the function itself. That's a reasonable grammatical inference, at least.

Why would you want to refer to a function from inside itself? The most common reasons would be things like recursion (calling a function from inside itself) or having an event handler that can unbind itself when it's first called.

Developers new to JS's mechanisms often think that referencing the function as an object (all functions in JavaScript are objects!) lets you store *state* (values in properties) between function calls. While this is certainly possible and has some limited uses, the rest of the book will expound on many other patterns for *better* places to store state besides the function object.

But for just a moment, we'll explore that pattern, to illustrate how `this` doesn't let a function get a reference to itself like we might have assumed.

Consider the following code, where we attempt to track how many times a function (`foo`) was called:

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 0 -- WTF?
```

`foo.count` is *still* `0`, even though the four `console.log` statements clearly indicate `foo(..)` was in fact called four times. The frustration stems from a *too literal* interpretation of what `this` (in `this.count++`) means.

When the code executes `foo.count = 0`, indeed it's adding a property `count` to the function object `foo`. But for the `this.count` reference inside of the function, `this` is not in fact pointing *at all* to that function object, and so even though the property names are the same, the root objects are different, and confusion ensues.

Note: A responsible developer *should* ask at this point, "If I was incrementing a `count` property but it wasn't the one I expected, which `count` *was* I incrementing?" In fact, were she to dig deeper, she would find that she had accidentally created a global variable `count` (see Chapter 2 for *how* that happened!), and it currently has the value `NaN`. Of course, once she identifies this peculiar outcome, she then has a whole other set of questions: "How was it global, and why did it end up `NaN` instead of some proper count value?" (see Chapter 2).

Instead of stopping at this point and digging into why the `this` reference doesn't seem to be behaving as *expected*, and answering those tough but important questions, many developers simply avoid the issue altogether, and hack toward some other solution, such as creating another object to hold the `count` property:

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    data.count++;
}

var data = {
    count: 0
};

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( data.count ); // 4
```

While it is true that this approach "solves" the problem, unfortunately it simply ignores the real problem -- lack of understanding what `this` means and how it works -- and instead falls back to the comfort zone of a more familiar mechanism: lexical scope.

Note: Lexical scope is a perfectly fine and useful mechanism; I am not belittling the use of it, by any means (see *"Scope & Closures"* title of this book series). But constantly *guessing* at how to use `this`, and usually being *wrong*, is not a good reason to retreat back to lexical scope and never learn *why* `this` eludes you.

To reference a function object from inside itself, `this` by itself will typically be insufficient. You generally need a reference to the function object via a lexical identifier (variable) that points at it.

Consider these two functions:

```
function foo() {
    foo.count = 4; // `foo` refers to itself
}

setTimeout( function(){
    // anonymous function (no name), cannot
    // refer to itself
}, 10 );
```

In the first function, called a "named function", `foo` is a reference that can be used to refer to the function from inside itself.

But in the second example, the function callback passed to `setTimeout(..)` has no name identifier (so called an "anonymous function"), so there's no proper way to refer to the function object itself.

Note: The old-school but now deprecated and frowned-upon `arguments.callee` reference inside a function *also* points to the function object of the currently executing function. This reference is typically the only way to access an anonymous function's object from inside itself. The best approach, however, is to avoid the use of anonymous functions altogether, at least for those which require a self-reference, and instead use a named function (expression). `arguments.callee` is deprecated and should not be used.

So another solution to our running example would have been to use the `foo` identifier as a function object reference in each place, and not use `this` at all, which *works*:

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    foo.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 4
```

However, that approach similarly side-steps *actual* understanding of `this` and relies entirely on the lexical scoping of variable `foo`.

Yet another way of approaching the issue is to force `this` to actually point at the `foo` function object:

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    // Note: `this` IS actually `foo` now, based on
    // how `foo` is called (see below)
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        // using `call(..)`, we ensure the `this`
        // points at the function object (`foo`) itself
        foo.call( foo, i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 4
```

Instead of avoiding `this`, we embrace it. We'll explain in a little bit *how* such techniques work much more completely, so don't worry if you're still a bit confused!

Its Scope

The next most common misconception about the meaning of `this` is that it somehow refers to the function's scope. It's a tricky question, because in one sense there is some truth, but in the other sense, it's quite misguided.

To be clear, `this` does not, in any way, refer to a function's **lexical scope**. It is true that internally, scope is kind of like an object with properties for each of the available identifiers. But the scope "object" is not accessible to JavaScript code. It's an inner part of the *Engine*'s implementation.

Consider code which attempts (and fails!) to cross over the boundary and use `this` to implicitly refer to a function's lexical scope:

```
function foo() {
    var a = 2;
    this.bar();
}

function bar() {
    console.log( this.a );
}

foo(); //undefined
```

There's more than one mistake in this snippet. While it may seem contrived, the code you see is a distillation of actual real-world code that has been exchanged in public community help forums. It's a wonderful (if not sad) illustration of just how misguided `this` assumptions can be.

Firstly, an attempt is made to reference the `bar()` function via `this.bar()`. It is almost certainly an *accident* that it works, but we'll explain the *how* of that shortly. The most natural way to have invoked `bar()` would have been to omit the leading `this.` and just make a lexical reference to the identifier.

However, the developer who writes such code is attempting to use `this` to create a bridge between the lexical scopes of `foo()` and `bar()`, so that `bar()` has access to the variable `a` in the inner scope of `foo()`. **No such bridge is possible.** You cannot use a `this` reference to look something up in a lexical scope. It is not possible.

Every time you feel yourself trying to mix lexical scope look-ups with `this`, remind yourself: *there is no bridge*.

What's `this`?

Having set aside various incorrect assumptions, let us now turn our attention to how the `this` mechanism really works.

We said earlier that `this` is not an author-time binding but a runtime binding. It is contextual based on the conditions of the function's invocation. `this` binding has nothing to do with where a function is declared, but has instead everything to do with the manner in which the function is called.

When a function is invoked, an activation record, otherwise known as an execution context, is created. This record contains information about where the function was called from (the call-stack), *how* the function was invoked, what parameters were passed, etc. One of the

properties of this record is the `this` reference which will be used for the duration of that function's execution.

In the next chapter, we will learn to find a function's **call-site** to determine how its execution will bind `this`.

Review (TL;DR)

`this` binding is a constant source of confusion for the JavaScript developer who does not take the time to learn how the mechanism actually works. Guesses, trial-and-error, and blind copy-n-paste from Stack Overflow answers is not an effective or proper way to leverage *this* important `this` mechanism.

To learn `this`, you first have to learn what `this` is *not*, despite any assumptions or misconceptions that may lead you down those paths. `this` is neither a reference to the function itself, nor is it a reference to the function's *lexical* scope.

`this` is actually a binding that is made when a function is invoked, and *what* it references is determined entirely by the call-site where the function is called.

You Don't Know JS: *this* & Object Prototypes

Chapter 2: `this` All Makes Sense Now!

In Chapter 1, we discarded various misconceptions about `this` and learned instead that `this` is a binding made for each function invocation, based entirely on its **call-site** (how the function is called).

Call-site

To understand `this` binding, we have to understand the call-site: the location in code where a function is called (**not where it's declared**). We must inspect the call-site to answer the question: what's *this* `this` a reference to?

Finding the call-site is generally: "go locate where a function is called from", but it's not always that easy, as certain coding patterns can obscure the *true* call-site.

What's important is to think about the **call-stack** (the stack of functions that have been called to get us to the current moment in execution). The call-site we care about is *in* the invocation *before* the currently executing function.

Let's demonstrate call-stack and call-site:

```
function baz() {
    // call-stack is: `baz`
    // so, our call-site is in the global scope

    console.log( "baz" );
    bar(); // <-- call-site for `bar`
}

function bar() {
    // call-stack is: `baz` -> `bar`
    // so, our call-site is in `baz`

    console.log( "bar" );
    foo(); // <-- call-site for `foo`
}

function foo() {
    // call-stack is: `baz` -> `bar` -> `foo`
    // so, our call-site is in `bar`

    console.log( "foo" );
}

baz(); // <-- call-site for `baz`
```

Take care when analyzing code to find the actual call-site (from the call-stack), because it's the only thing that matters for `this` binding.

Note: You can visualize a call-stack in your mind by looking at the chain of function calls in order, as we did with the comments in the above snippet. But this is painstaking and error-prone. Another way of seeing the call-stack is using a debugger tool in your browser. Most modern desktop browsers have built-in developer tools, which includes a JS debugger. In the above snippet, you could have set a breakpoint in the tools for the first line of the `foo()` function, or simply inserted the `debugger;` statement on that first line. When you run the page, the debugger will pause at this location, and will show you a list of the functions that have been called to get to that line, which will be your call stack. So, if you're trying to diagnose `this` binding, use the developer tools to get the call-stack, then find the second item from the top, and that will show you the real call-site.

Nothing But Rules

We turn our attention now to *how* the call-site determines where `this` will point during the execution of a function.

You must inspect the call-site and determine which of 4 rules applies. We will first explain each of these 4 rules independently, and then we will illustrate their order of precedence, if multiple rules *could* apply to the call-site.

Default Binding

The first rule we will examine comes from the most common case of function calls: standalone function invocation. Think of *this* `this` rule as the default catch-all rule when none of the other rules apply.

Consider this code:

```
function foo() {
    console.log( this.a );
}

var a = 2;

foo(); // 2
```

The first thing to note, if you were not already aware, is that variables declared in the global scope, as `var a = 2` is, are synonymous with global-object properties of the same name. They're not copies of each other, they *are* each other. Think of it as two sides of the same coin.

Secondly, we see that when `foo()` is called, `this.a` resolves to our global variable `a`. Why? Because in this case, the *default binding* for `this` applies to the function call, and so points `this` at the global object.

How do we know that the *default binding* rule applies here? We examine the call-site to see how `foo()` is called. In our snippet, `foo()` is called with a plain, un-decorated function reference. None of the other rules we will demonstrate will apply here, so the *default binding* applies instead.

If `strict mode` is in effect, the global object is not eligible for the *default binding*, so the `this` is instead set to `undefined`.

```
function foo() {
    "use strict";

    console.log( this.a );
}

var a = 2;

foo(); // TypeError: `this` is `undefined`
```

A subtle but important detail is: even though the overall `this` binding rules are entirely based on the call-site, the global object is **only** eligible for the *default binding* if the **contents** of `foo()` are **not** running in `strict mode`; the `strict mode` state of the call-site of `foo()` is irrelevant.

```
function foo() {
    console.log( this.a );
}

var a = 2;

(function(){
    "use strict";

    foo(); // 2
})();
```

Note: Intentionally mixing `strict mode` and non-`strict mode` together in your own code is generally frowned upon. Your entire program should probably either be **Strict** or **non-Strict**. However, sometimes you include a third-party library that has different **Strict**'ness than your own code, so care must be taken over these subtle compatibility details.

Implicit Binding

Another rule to consider is: does the call-site have a context object, also referred to as an owning or containing object, though *these* alternate terms could be slightly misleading.

Consider:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

Firstly, notice the manner in which `foo()` is declared and then later added as a reference property onto `obj`. Regardless of whether `foo()` is initially declared *on* `obj`, or is added as a reference later (as this snippet shows), in neither case is the **function** really "owned" or "contained" by the `obj` object.

However, the call-site *uses* the `obj` context to **reference** the function, so you *could* say that the `obj` object "owns" or "contains" the **function reference** at the time the function is called.

Whatever you choose to call this pattern, at the point that `foo()` is called, it's preceded by an object reference to `obj`. When there is a context object for a function reference, the *implicit binding* rule says that it's *that* object which should be used for the function call's `this` binding.

Because `obj` is the `this` for the `foo()` call, `this.a` is synonymous with `obj.a`.

Only the top/last level of an object property reference chain matters to the call-site. For instance:

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

Implicitly Lost

One of the most common frustrations that `this` binding creates is when an *implicitly bound* function loses that binding, which usually means it falls back to the *default binding*, of either the global object or `undefined`, depending on `strict mode`.

Consider:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo; // function reference/alias!

var a = "oops, global"; // `a` also property on global object

bar(); // "oops, global"
```

Even though `bar` appears to be a reference to `obj.foo`, in fact, it's really just another reference to `foo` itself. Moreover, the call-site is what matters, and the call-site is `bar()`, which is a plain, un-decorated call and thus the *default binding* applies.

The more subtle, more common, and more unexpected way this occurs is when we consider passing a callback function:

```
function foo() {
    console.log( this.a );
}

function doFoo(fn) {
    // `fn` is just another reference to `foo`

    fn(); // <-- call-site!
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a` also property on global object

doFoo( obj.foo ); // "oops, global"
```

Parameter passing is just an implicit assignment, and since we're passing a function, it's an implicit reference assignment, so the end result is the same as the previous snippet.

What if the function you're passing your callback to is not your own, but built-in to the language? No difference, same outcome.

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a` also property on global object

setTimeout( obj.foo, 100 ); // "oops, global"
```

Think about this crude theoretical pseudo-implementation of `setTimeout()` provided as a built-in from the JavaScript environment:

```
function setTimeout(fn,delay) {
    // wait (somehow) for `delay` milliseconds
    fn(); // <-- call-site!
}
```

It's quite common that our function callbacks *lose* their `this` binding, as we've just seen. But another way that `this` can surprise us is when the function we've passed our callback to intentionally changes the `this` for the call. Event handlers in popular JavaScript libraries are quite fond of forcing your callback to have a `this` which points to, for instance, the DOM element that triggered the event. While that may sometimes be useful, other times it can be downright infuriating. Unfortunately, these tools rarely let you choose.

Either way the `this` is changed unexpectedly, you are not really in control of how your callback function reference will be executed, so you have no way (yet) of controlling the call-site to give your intended binding. We'll see shortly a way of "fixing" that problem by *fixing* the `this`.

Explicit Binding

With *implicit binding* as we just saw, we had to mutate the object in question to include a reference on itself to the function, and use this property function reference to indirectly (implicitly) bind `this` to the object.

But, what if you want to force a function call to use a particular object for the `this` binding, without putting a property function reference on the object?

"All" functions in the language have some utilities available to them (via their `[[Prototype]]` -- more on that later) which can be useful for this task. Specifically, functions have `call(..)` and `apply(..)` methods. Technically, JavaScript host environments sometimes provide functions which are special enough (a kind way of putting it!) that they do not have such functionality. But those are few. The vast majority of functions provided, and certainly all functions you will create, do have access to `call(..)` and `apply(..)`.

How do these utilities work? They both take, as their first parameter, an object to use for the `this`, and then invoke the function with that `this` specified. Since you are directly stating what you want the `this` to be, we call it *explicit binding*.

Consider:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Invoking `foo` with *explicit binding* by `foo.call(..)` allows us to force its `this` to be `obj`.

If you pass a simple primitive value (of type `string`, `boolean`, or `number`) as the `this` binding, the primitive value is wrapped in its object-form (`new String(..)`, `new Boolean(..)`, or `new Number(..)`, respectively). This is often referred to as "boxing".

Note: With respect to `this` binding, `call(..)` and `apply(..)` are identical. They *do* behave differently with their additional parameters, but that's not something we care about presently.

Unfortunately, *explicit binding* alone still doesn't offer any solution to the issue mentioned previously, of a function "losing" its intended `this` binding, or just having it paved over by a framework, etc.

Hard Binding

But a variation pattern around *explicit binding* actually does the trick. Consider:

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

var bar = function() {
    foo.call( obj );
};

bar(); // 2
setTimeout( bar, 100 ); // 2

// `bar` hard binds `foo`'s `this` to `obj`
// so that it cannot be overriden
bar.call( window ); // 2
```

Let's examine how this variation works. We create a function `bar()` which, internally, manually calls `foo.call(obj)`, thereby forcibly invoking `foo` with `obj` binding for `this`. No matter how you later invoke the function `bar`, it will always manually invoke `foo` with `obj`. This binding is both explicit and strong, so we call it *hard binding*.

The most typical way to wrap a function with a *hard binding* creates a pass-thru of any arguments passed and any return value received:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = function() {
    return foo.apply( obj, arguments );
};

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Another way to express this pattern is to create a re-usable helper:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

// simple `bind` helper
function bind(fn, obj) {
    return function() {
        return fn.apply( obj, arguments );
    };
}

var obj = {
    a: 2
};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Since *hard binding* is such a common pattern, it's provided with a built-in utility as of ES5: `Function.prototype.bind` , and it's used like this:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

`bind(..)` returns a new function that is hard-coded to call the original function with the `this` context set as you specified.

Note: As of ES6, the hard-bound function produced by `bind(..)` has a `.name` property that derives from the original *target function*. For example: `bar = foo.bind(..)` should have a `bar.name` value of `"bound foo"` , which is the function call name that should show up in a stack trace.

API Call "Contexts"

Many libraries' functions, and indeed many new built-in functions in the JavaScript language and host environment, provide an optional parameter, usually called "context", which is designed as a work-around for you not having to use `bind(..)` to ensure your callback function uses a particular `this`.

For instance:

```
function foo(el) {
	console.log( el, this.id );
}

var obj = {
	id: "awesome"
};

// use `obj` as `this` for `foo(..)` calls
[1, 2, 3].forEach( foo, obj ); // 1 awesome  2 awesome  3 awesome
```

Internally, these various functions almost certainly use *explicit binding* via `call(..)` or `apply(..)`, saving you the trouble.

`new` Binding

The fourth and final rule for `this` binding requires us to re-think a very common misconception about functions and objects in JavaScript.

In traditional class-oriented languages, "constructors" are special methods attached to classes, that when the class is instantiated with a `new` operator, the constructor of that class is called. This usually looks something like:

```
something = new MyClass(..);
```

JavaScript has a `new` operator, and the code pattern to use it looks basically identical to what we see in those class-oriented languages; most developers assume that JavaScript's mechanism is doing something similar. However, there really is *no connection* to class-oriented functionality implied by `new` usage in JS.

First, let's re-define what a "constructor" in JavaScript is. In JS, constructors are **just functions** that happen to be called with the `new` operator in front of them. They are not attached to classes, nor are they instantiating a class. They are not even special types of functions. They're just regular functions that are, in essence, hijacked by the use of `new` in their invocation.

For example, the `Number(..)` function acting as a constructor, quoting from the ES5.1 spec:

> 15.7.2 The Number Constructor
>
> When Number is called as part of a new expression it is a constructor: it initialises the newly created object.

So, pretty much any ol' function, including the built-in object functions like `Number(..)` (see Chapter 3) can be called with `new` in front of it, and that makes that function call a *constructor call*. This is an important but subtle distinction: there's really no such thing as "constructor functions", but rather construction calls *of* functions.

When a function is invoked with `new` in front of it, otherwise known as a constructor call, the following things are done automatically:

1. a brand new object is created (aka, constructed) out of thin air
2. *the newly constructed object is `[[Prototype]]` -linked*
3. the newly constructed object is set as the `this` binding for that function call
4. unless the function returns its own alternate **object**, the `new` -invoked function call will *automatically* return the newly constructed object.

Steps 1, 3, and 4 apply to our current discussion. We'll skip over step 2 for now and come back to it in Chapter 5.

Consider this code:

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

By calling `foo(..)` with `new` in front of it, we've constructed a new object and set that new object as the `this` for the call of `foo(..)`. **So `new` is the final way that a function call's `this` can be bound.** We'll call this *new binding*.

Everything In Order

So, now we've uncovered the 4 rules for binding `this` in function calls. *All* you need to do is find the call-site and inspect it to see which rule applies. But, what if the call-site has multiple eligible rules? There must be an order of precedence to these rules, and so we will next demonstrate what order to apply the rules.

It should be clear that the *default binding* is the lowest priority rule of the 4. So we'll just set that one aside.

Which is more precedent, *implicit binding* or *explicit binding*? Let's test it:

```
function foo() {
    console.log( this.a );
}

var obj1 = {
    a: 2,
    foo: foo
};

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

So, *explicit binding* takes precedence over *implicit binding*, which means you should ask **first** if *explicit binding* applies before checking for *implicit binding*.

Now, we just need to figure out where *new binding* fits in the precedence.

```
function foo(something) {
    this.a = something;
}

var obj1 = {
    foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
console.log( bar.a ); // 4
```

OK, *new binding* is more precedent than *implicit binding*. But do you think *new binding* is more or less precedent than *explicit binding*?

Note: `new` and `call` / `apply` cannot be used together, so `new foo.call(obj1)` is not allowed, to test *new binding* directly against *explicit binding*. But we can still use a *hard binding* to test the precedence of the two rules.

Before we explore that in a code listing, think back to how *hard binding* physically works, which is that `Function.prototype.bind(..)` creates a new wrapper function that is hard-coded to ignore its own `this` binding (whatever it may be), and use a manual one we provide.

By that reasoning, it would seem obvious to assume that *hard binding* (which is a form of *explicit binding*) is more precedent than *new binding*, and thus cannot be overridden with `new`.

Let's check:

```
function foo(something) {
    this.a = something;
}

var obj1 = {};

var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar( 3 );
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

Whoa! `bar` is hard-bound against `obj1`, but `new bar(3)` did **not** change `obj1.a` to be `3` as we would have expected. Instead, the *hard bound* (to `obj1`) call to `bar(..)` ***is*** able to be overridden with `new`. Since `new` was applied, we got the newly created object back, which we named `baz`, and we see in fact that `baz.a` has the value `3`.

This should be surprising if you go back to our "fake" bind helper:

```
function bind(fn, obj) {
    return function() {
        fn.apply( obj, arguments );
    };
}
```

If you reason about how the helper's code works, it does not have a way for a `new` operator call to override the hard-binding to `obj` as we just observed.

But the built-in `Function.prototype.bind(..)` as of ES5 is more sophisticated, quite a bit so in fact. Here is the (slightly reformatted) polyfill provided by the MDN page for `bind(..)`:

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(oThis) {
        if (typeof this !== "function") {
            // closest thing possible to the ECMAScript 5
            // internal IsCallable function
            throw new TypeError( "Function.prototype.bind - what " +
                "is trying to be bound is not callable"
            );
        }

        var aArgs = Array.prototype.slice.call( arguments, 1 ),
            fToBind = this,
            fNOP = function(){},
            fBound = function(){
                return fToBind.apply(
                    (
                        this instanceof fNOP &&
                        oThis ? this : oThis
                    ),
                    aArgs.concat( Array.prototype.slice.call( arguments ) )
                );
            }
        ;

        fNOP.prototype = this.prototype;
        fBound.prototype = new fNOP();

        return fBound;
    };
}
```

Note: The `bind(..)` polyfill shown above differs from the built-in `bind(..)` in ES5 with respect to hard-bound functions that will be used with `new` (see below for why that's useful). Because the polyfill cannot create a function without a `.prototype` as the built-in utility does, there's some nuanced indirection to approximate the same behavior. Tread carefully if you plan to use `new` with a hard-bound function and you rely on this polyfill.

The part that's allowing `new` overriding is:

```
this instanceof fNOP &&
oThis ? this : oThis

// ... and:

fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();
```

We won't actually dive into explaining how this trickery works (it's complicated and beyond our scope here), but essentially the utility determines whether or not the hard-bound function has been called with `new` (resulting in a newly constructed object being its `this`), and if so, it uses *that* newly created `this` rather than the previously specified *hard binding* for `this`.

Why is `new` being able to override *hard binding* useful?

The primary reason for this behavior is to create a function (that can be used with `new` for constructing objects) that essentially ignores the `this` *hard binding* but which presets some or all of the function's arguments. One of the capabilities of `bind(..)` is that any arguments passed after the first `this` binding argument are defaulted as standard arguments to the underlying function (technically called "partial application", which is a subset of "currying").

For example:

```
function foo(p1,p2) {
    this.val = p1 + p2;
}

// using `null` here because we don't care about
// the `this` hard-binding in this scenario, and
// it will be overridden by the `new` call anyway!
var bar = foo.bind( null, "p1" );

var baz = new bar( "p2" );

baz.val; // p1p2
```

Determining `this`

Now, we can summarize the rules for determining `this` from a function call's call-site, in their order of precedence. Ask these questions in this order, and stop when the first rule applies.

1. Is the function called with `new` (**new binding**)? If so, `this` is the newly constructed object.

   `var bar = new foo()`

2. Is the function called with `call` or `apply` (**explicit binding**), even hidden inside a `bind` *hard binding*? If so, `this` is the explicitly specified object.

   `var bar = foo.call( obj2 )`

3. Is the function called with a context (**implicit binding**), otherwise known as an owning or containing object? If so, `this` is *that* context object.

    `var bar = obj1.foo()`

4. Otherwise, default the `this` (**default binding**). If in `strict mode`, pick `undefined`, otherwise pick the `global` object.

    `var bar = foo()`

That's it. That's *all it takes* to understand the rules of `this` binding for normal function calls. Well... almost.

Binding Exceptions

As usual, there are some *exceptions* to the "rules".

The `this`-binding behavior can in some scenarios be surprising, where you intended a different binding but you end up with binding behavior from the *default binding* rule (see previous).

Ignored `this`

If you pass `null` or `undefined` as a `this` binding parameter to `call`, `apply`, or `bind`, those values are effectively ignored, and instead the *default binding* rule applies to the invocation.

```
function foo() {
    console.log( this.a );
}

var a = 2;

foo.call( null ); // 2
```

Why would you intentionally pass something like `null` for a `this` binding?

It's quite common to use `apply(..)` for spreading out arrays of values as parameters to a function call. Similarly, `bind(..)` can curry parameters (pre-set values), which can be very helpful.

```
function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}

// spreading out array as parameters
foo.apply( null, [2, 3] ); // a:2, b:3

// currying with `bind(..)`
var bar = foo.bind( null, 2 );
bar( 3 ); // a:2, b:3
```

Both these utilities require a `this` binding for the first parameter. If the functions in question don't care about `this` , you need a placeholder value, and `null` might seem like a reasonable choice as shown in this snippet.

Note: We don't cover it in this book, but ES6 has the `...` spread operator which will let you syntactically "spread out" an array as parameters without needing `apply(..)` , such as `foo(...[1,2])` , which amounts to `foo(1,2)` -- syntactically avoiding a `this` binding if it's unnecessary. Unfortunately, there's no ES6 syntactic substitute for currying, so the `this` parameter of the `bind(..)` call still needs attention.

However, there's a slight hidden "danger" in always using `null` when you don't care about the `this` binding. If you ever use that against a function call (for instance, a third-party library function that you don't control), and that function *does* make a `this` reference, the *default binding* rule means it might inadvertently reference (or worse, mutate!) the `global` object (`window` in the browser).

Obviously, such a pitfall can lead to a variety of *very difficult* to diagnose/track-down bugs.

Safer `this`

Perhaps a somewhat "safer" practice is to pass a specifically set up object for `this` which is guaranteed not to be an object that can create problematic side effects in your program. Borrowing terminology from networking (and the military), we can create a "DMZ" (de-militarized zone) object -- nothing more special than a completely empty, non-delegated (see Chapters 5 and 6) object.

If we always pass a DMZ object for ignored `this` bindings we don't think we need to care about, we're sure any hidden/unexpected usage of `this` will be restricted to the empty object, which insulates our program's `global` object from side-effects.

Since this object is totally empty, I personally like to give it the variable name `ø` (the lowercase mathematical symbol for the empty set). On many keyboards (like US-layout on Mac), this symbol is easily typed with `⌥` + `o` (option+ `o`). Some systems also let you set

up hotkeys for specific symbols. If you don't like the `ø` symbol, or your keyboard doesn't make that as easy to type, you can of course call it whatever you want.

Whatever you call it, the easiest way to set it up as **totally empty** is `Object.create(null)` (see Chapter 5). `Object.create(null)` is similar to `{ }`, but without the delegation to `Object.prototype`, so it's "more empty" than just `{ }`.

```
function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}

// our DMZ empty object
var ø = Object.create( null );

// spreading out array as parameters
foo.apply( ø, [2, 3] ); // a:2, b:3

// currying with `bind(..)`
var bar = foo.bind( ø, 2 );
bar( 3 ); // a:2, b:3
```

Not only functionally "safer", there's a sort of stylistic benefit to `ø`, in that it semantically conveys "I want the `this` to be empty" a little more clearly than `null` might. But again, name your DMZ object whatever you prefer.

Indirection

Another thing to be aware of is you can (intentionally or not!) create "indirect references" to functions, and in those cases, when that function reference is invoked, the *default binding* rule also applies.

One of the most common ways that *indirect references* occur is from an assignment:

```
function foo() {
    console.log( this.a );
}

var a = 2;
var o = { a: 3, foo: foo };
var p = { a: 4 };

o.foo(); // 3
(p.foo = o.foo)(); // 2
```

The *result value* of the assignment expression `p.foo = o.foo` is a reference to just the underlying function object. As such, the effective call-site is just `foo()`, not `p.foo()` or `o.foo()` as you might expect. Per the rules above, the *default binding* rule applies.

Reminder: regardless of how you get to a function invocation using the *default binding* rule, the `strict mode` status of the **contents** of the invoked function making the `this` reference -- not the function call-site -- determines the *default binding* value: either the `global` object if in non-`strict mode` or `undefined` if in `strict mode`.

Softening Binding

We saw earlier that *hard binding* was one strategy for preventing a function call falling back to the *default binding* rule inadvertently, by forcing it to be bound to a specific `this` (unless you use `new` to override it!). The problem is, *hard-binding* greatly reduces the flexibility of a function, preventing manual `this` override with either the *implicit binding* or even subsequent *explicit binding* attempts.

It would be nice if there was a way to provide a different default for *default binding* (not `global` or `undefined`), while still leaving the function able to be manually `this` bound via *implicit binding* or *explicit binding* techniques.

We can construct a so-called *soft binding* utility which emulates our desired behavior.

```
if (!Function.prototype.softBind) {
    Function.prototype.softBind = function(obj) {
        var fn = this,
            curried = [].slice.call( arguments, 1 ),
            bound = function bound() {
                return fn.apply(
                    (!this ||
                        (typeof window !== "undefined" &&
                            this === window) ||
                        (typeof global !== "undefined" &&
                            this === global)
                    ) ? obj : this,
                    curried.concat.apply( curried, arguments )
                );
            };
        bound.prototype = Object.create( fn.prototype );
        return bound;
    };
}
```

The `softBind(..)` utility provided here works similarly to the built-in ES5 `bind(..)` utility, except with our *soft binding* behavior. It wraps the specified function in logic that checks the `this` at call-time and if it's `global` or `undefined`, uses a pre-specified alternate *default*

(`obj`). Otherwise the `this` is left untouched. It also provides optional currying (see the `bind(..)` discussion earlier).

Let's demonstrate its usage:

```
function foo() {
   console.log("name: " + this.name);
}

var obj = { name: "obj" },
    obj2 = { name: "obj2" },
    obj3 = { name: "obj3" };

var fooOBJ = foo.softBind( obj );

fooOBJ(); // name: obj

obj2.foo = foo.softBind(obj);
obj2.foo(); // name: obj2   <---- look!!!

fooOBJ.call( obj3 ); // name: obj3   <---- look!

setTimeout( obj2.foo, 10 ); // name: obj   <---- falls back to soft-binding
```

The soft-bound version of the `foo()` function can be manually `this` -bound to `obj2` or `obj3` as shown, but it falls back to `obj` if the *default binding* would otherwise apply.

Lexical `this`

Normal functions abide by the 4 rules we just covered. But ES6 introduces a special kind of function that does not use these rules: arrow-function.

Arrow-functions are signified not by the `function` keyword, but by the `=>` so called "fat arrow" operator. Instead of using the four standard `this` rules, arrow-functions adopt the `this` binding from the enclosing (function or global) scope.

Let's illustrate arrow-function lexical scope:

```
function foo() {
    // return an arrow function
    return (a) => {
        // `this` here is lexically adopted from `foo()`
        console.log( this.a );
    };
}

var obj1 = {
    a: 2
};

var obj2 = {
    a: 3
};

var bar = foo.call( obj1 );
bar.call( obj2 ); // 2, not 3!
```

The arrow-function created in `foo()` lexically captures whatever `foo()` s `this` is at its call-time. Since `foo()` was `this` -bound to `obj1` , `bar` (a reference to the returned arrow-function) will also be `this` -bound to `obj1` . The lexical binding of an arrow-function cannot be overridden (even with `new` !).

The most common use-case will likely be in the use of callbacks, such as event handlers or timers:

```
function foo() {
    setTimeout(() => {
        // `this` here is lexically adopted from `foo()`
        console.log( this.a );
    },100);
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

While arrow-functions provide an alternative to using `bind(..)` on a function to ensure its `this` , which can seem attractive, it's important to note that they essentially are disabling the traditional `this` mechanism in favor of more widely-understood lexical scoping. Pre-ES6, we already have a fairly common pattern for doing so, which is basically almost indistinguishable from the spirit of ES6 arrow-functions:

```
function foo() {
    var self = this; // lexical capture of `this`
    setTimeout( function(){
        console.log( self.a );
    }, 100 );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

While `self = this` and arrow-functions both seem like good "solutions" to not wanting to use `bind(..)`, they are essentially fleeing from `this` instead of understanding and embracing it.

If you find yourself writing `this`-style code, but most or all the time, you defeat the `this` mechanism with lexical `self = this` or arrow-function "tricks", perhaps you should either:

1. Use only lexical scope and forget the false pretense of `this`-style code.
2. Embrace `this`-style mechanisms completely, including using `bind(..)` where necessary, and try to avoid `self = this` and arrow-function "lexical this" tricks.

A program can effectively use both styles of code (lexical and `this`), but inside of the same function, and indeed for the same sorts of look-ups, mixing the two mechanisms is usually asking for harder-to-maintain code, and probably working too hard to be clever.

Review (TL;DR)

Determining the `this` binding for an executing function requires finding the direct call-site of that function. Once examined, four rules can be applied to the call-site, in *this* order of precedence:

1. Called with `new`? Use the newly constructed object.
2. Called with `call` or `apply` (or `bind`)? Use the specified object.
3. Called with a context object owning the call? Use that context object.
4. Default: `undefined` in `strict mode`, global object otherwise.

Be careful of accidental/unintentional invoking of the *default binding* rule. In cases where you want to "safely" ignore a `this` binding, a "DMZ" object like `ø = Object.create(null)` is a good placeholder value that protects the `global` object from unintended side-effects.

Instead of the four standard binding rules, ES6 arrow-functions use lexical scoping for `this` binding, which means they adopt the `this` binding (whatever it is) from its enclosing function call. They are essentially a syntactic replacement of `self = this` in pre-ES6 coding.

You Don't Know JS: *this* & Object Prototypes

Chapter 3: Objects

In Chapters 1 and 2, we explained how the `this` binding points to various objects depending on the call-site of the function invocation. But what exactly are objects, and why do we need to point to them? We will explore objects in detail in this chapter.

Syntax

Objects come in two forms: the declarative (literal) form, and the constructed form.

The literal syntax for an object looks like this:

```
var myObj = {
    key: value
    // ...
};
```

The constructed form looks like this:

```
var myObj = new Object();
myObj.key = value;
```

The constructed form and the literal form result in exactly the same sort of object. The only difference really is that you can add one or more key/value pairs to the literal declaration, whereas with constructed-form objects, you must add the properties one-by-one.

Note: It's extremely uncommon to use the "constructed form" for creating objects as just shown. You would pretty much always want to use the literal syntax form. The same will be true of most of the built-in objects (see below).

Type

Objects are the general building block upon which much of JS is built. They are one of the 6 primary types (called "language types" in the specification) in JS:

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `object`

Note that the *simple primitives* (`string` , `number` , `boolean` , `null` , and `undefined`) are **not** themselves `objects` . `null` is sometimes referred to as an object type, but this misconception stems from a bug in the language which causes `typeof null` to return the string `"object"` incorrectly (and confusingly). In fact, `null` is its own primitive type.

It's a common mis-statement that "everything in JavaScript is an object". This is clearly not true.

By contrast, there *are* a few special object sub-types, which we can refer to as *complex primitives*.

`function` is a sub-type of object (technically, a "callable object"). Functions in JS are said to be "first class" in that they are basically just normal objects (with callable behavior semantics bolted on), and so they can be handled like any other plain object.

Arrays are also a form of objects, with extra behavior. The organization of contents in arrays is slightly more structured than for general objects.

Built-in Objects

There are several other object sub-types, usually referred to as built-in objects. For some of them, their names seem to imply they are directly related to their simple primitives counter-parts, but in fact, their relationship is more complicated, which we'll explore shortly.

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`
- `Date`
- `RegExp`
- `Error`

These built-ins have the appearance of being actual types, even classes, if you rely on the similarity to other languages such as Java's `String` class.

But in JS, these are actually just built-in functions. Each of these built-in functions can be used as a constructor (that is, a function call with the `new` operator -- see Chapter 2), with the result being a newly *constructed* object of the sub-type in question. For instance:

```
var strPrimitive = "I am a string";
typeof strPrimitive;                          // "string"
strPrimitive instanceof String;                  // false

var strObject = new String( "I am a string" );
typeof strObject;                                // "object"
strObject instanceof String;                  // true

// inspect the object sub-type
Object.prototype.toString.call( strObject );    // [object String]
```

We'll see in detail in a later chapter exactly how the `Object.prototype.toString...` bit works, but briefly, we can inspect the internal sub-type by borrowing the base default `toString()` method, and you can see it reveals that `strObject` is an object that was in fact created by the `String` constructor.

The primitive value `"I am a string"` is not an object, it's a primitive literal and immutable value. To perform operations on it, such as checking its length, accessing its individual character contents, etc, a `String` object is required.

Luckily, the language automatically coerces a `"string"` primitive to a `String` object when necessary, which means you almost never need to explicitly create the Object form. It is **strongly preferred** by the majority of the JS community to use the literal form for a value, where possible, rather than the constructed object form.

Consider:

```
var strPrimitive = "I am a string";

console.log( strPrimitive.length );            // 13

console.log( strPrimitive.charAt( 3 ) );    // "m"
```

In both cases, we call a property or method on a string primitive, and the engine automatically coerces it to a `String` object, so that the property/method access works.

The same sort of coercion happens between the number literal primitive `42` and the `new Number(42)` object wrapper, when using methods like `42.359.toFixed(2)`. Likewise for `Boolean` objects from `"boolean"` primitives.

`null` and `undefined` have no object wrapper form, only their primitive values. By contrast, `Date` values can *only* be created with their constructed object form, as they have no literal form counter-part.

`Object` s, `Array` s, `Function` s, and `RegExp` s (regular expressions) are all objects regardless of whether the literal or constructed form is used. The constructed form does offer, in some cases, more options in creation than the literal form counterpart. Since objects are created either way, the simpler literal form is almost universally preferred. **Only use the constructed form if you need the extra options.**

`Error` objects are rarely created explicitly in code, but usually created automatically when exceptions are thrown. They can be created with the constructed form `new Error(..)` , but it's often unnecessary.

Contents

As mentioned earlier, the contents of an object consist of values (any type) stored at specifically named *locations*, which we call properties.

It's important to note that while we say "contents" which implies that these values are *actually* stored inside the object, that's merely an appearance. The engine stores values in implementation-dependent ways, and may very well not store them *in* some object container. What *is* stored in the container are these property names, which act as pointers (technically, *references*) to where the values are stored.

Consider:

```
var myObject = {
    a: 2
};

myObject.a;         // 2

myObject["a"];    // 2
```

To access the value at the *location* `a` in `myObject` , we need to use either the `.` operator or the `[ ]` operator. The `.a` syntax is usually referred to as "property" access, whereas the `["a"]` syntax is usually referred to as "key" access. In reality, they both access the same *location*, and will pull out the same value, `2` , so the terms can be used interchangeably. We will use the most common term, "property access" from here on.

The main difference between the two syntaxes is that the `.` operator requires an `Identifier` compatible property name after it, whereas the `[".."]` syntax can take basically any UTF-8/unicode compatible string as the name for the property. To reference a property of the name "Super-Fun!", for instance, you would have to use the `["Super-Fun!"]` access syntax, as `Super-Fun!` is not a valid `Identifier` property name.

Also, since the `[".."]` syntax uses a string's **value** to specify the location, this means the program can programmatically build up the value of the string, such as:

```
var wantA = true;
var myObject = {
    a: 2
};

var idx;

if (wantA) {
    idx = "a";
}

// later

console.log( myObject[idx] ); // 2
```

In objects, property names are **always** strings. If you use any other value besides a `string` (primitive) as the property, it will first be converted to a string. This even includes numbers, which are commonly used as array indexes, so be careful not to confuse the use of numbers between objects and arrays.

```
var myObject = { };

myObject[true] = "foo";
myObject[3] = "bar";
myObject[myObject] = "baz";

myObject["true"];                // "foo"
myObject["3"];                    // "bar"
myObject["[object Object]"];    // "baz"
```

Computed Property Names

The `myObject[..]` property access syntax we just described is useful if you need to use a computed expression value *as* the key name, like `myObject[prefix + name]`. But that's not really helpful when declaring objects using the object-literal syntax.

ES6 adds *computed property names*, where you can specify an expression, surrounded by a `[ ]` pair, in the key-name position of an object-literal declaration:

```
var prefix = "foo";

var myObject = {
    [prefix + "bar"]: "hello",
    [prefix + "baz"]: "world"
};

myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

The most common usage of *computed property names* will probably be for ES6 `Symbol` s, which we will not be covering in detail in this book. In short, they're a new primitive data type which has an opaque unguessable value (technically a `string` value). You will be strongly discouraged from working with the *actual value* of a `Symbol` (which can theoretically be different between different JS engines), so the name of the `Symbol` , like `Symbol.Something` (just a made up name!), will be what you use:

```
var myObject = {
    [Symbol.Something]: "hello world"
};
```

Property vs. Method

Some developers like to make a distinction when talking about a property access on an object, if the value being accessed happens to be a function. Because it's tempting to think of the function as *belonging* to the object, and in other languages, functions which belong to objects (aka, "classes") are referred to as "methods", it's not uncommon to hear, "method access" as opposed to "property access".

The specification makes this same distinction, interestingly.

Technically, functions never "belong" to objects, so saying that a function that just happens to be accessed on an object reference is automatically a "method" seems a bit of a stretch of semantics.

It *is* true that some functions have `this` references in them, and that *sometimes* these `this` references refer to the object reference at the call-site. But this usage really does not make that function any more a "method" than any other function, as `this` is dynamically bound at run-time, at the call-site, and thus its relationship to the object is indirect, at best.

Every time you access a property on an object, that is a **property access**, regardless of the type of value you get back. If you *happen* to get a function from that property access, it's not magically a "method" at that point. There's nothing special (outside of possible implicit `this` binding as explained earlier) about a function that comes from a property access.

For instance:

```
function foo() {
    console.log( "foo" );
}

var someFoo = foo;    // variable reference to `foo`

var myObject = {
    someFoo: foo
};

foo;                // function foo(){..}

someFoo;            // function foo(){..}

myObject.someFoo;    // function foo(){..}
```

`someFoo` and `myObject.someFoo` are just two separate references to the same function, and neither implies anything about the function being special or "owned" by any other object. If `foo()` above was defined to have a `this` reference inside it, that `myObject.someFoo` *implicit binding* would be the **only** observable difference between the two references. Neither reference really makes sense to be called a "method".

Perhaps one could argue that a function *becomes a method*, not at definition time, but during run-time just for that invocation, depending on how it's called at its call-site (with an object reference context or not -- see Chapter 2 for more details). Even this interpretation is a bit of a stretch.

The safest conclusion is probably that "function" and "method" are interchangeable in JavaScript.

Note: ES6 adds a `super` reference, which is typically going to be used with `class` (see Appendix A). The way `super` behaves (static binding rather than late binding as `this`) gives further weight to the idea that a function which is `super` bound somewhere is more a "method" than "function". But again, these are just subtle semantic (and mechanical) nuances.

Even when you declare a function expression as part of the object-literal, that function doesn't magically *belong* more to the object -- still just multiple references to the same function object:

```
var myObject = {
    foo: function foo() {
        console.log( "foo" );
    }
};

var someFoo = myObject.foo;

someFoo;        // function foo(){..}

myObject.foo;    // function foo(){..}
```

Note: In Chapter 6, we will cover an ES6 short-hand for that `foo: function foo(){ .. }` declaration syntax in our object-literal.

Arrays

Arrays also use the `[ ]` access form, but as mentioned above, they have slightly more structured organization for how and where values are stored (though still no restriction on what *type* of values are stored). Arrays assume *numeric indexing*, which means that values are stored in locations, usually called *indices*, at non-negative integers, such as `0` and `42`.

```
var myArray = [ "foo", 42, "bar" ];

myArray.length;        // 3

myArray[0];            // "foo"

myArray[2];            // "bar"
```

Arrays *are* objects, so even though each index is a positive integer, you can *also* add properties onto the array:

```
var myArray = [ "foo", 42, "bar" ];

myArray.baz = "baz";

myArray.length;    // 3

myArray.baz;    // "baz"
```

Notice that adding named properties (regardless of `.` or `[ ]` operator syntax) does not change the reported `length` of the array.

You *could* use an array as a plain key/value object, and never add any numeric indices, but this is a bad idea because arrays have behavior and optimizations specific to their intended use, and likewise with plain objects. Use objects to store key/value pairs, and arrays to store values at numeric indices.

Be careful: If you try to add a property to an array, but the property name *looks* like a number, it will end up instead as a numeric index (thus modifying the array contents):

```
var myArray = [ "foo", 42, "bar" ];

myArray["3"] = "baz";

myArray.length;    // 4

myArray[3];        // "baz"
```

Duplicating Objects

One of the most commonly requested features when developers newly take up the JavaScript language is how to duplicate an object. It would seem like there should just be a built-in `copy()` method, right? It turns out that it's a little more complicated than that, because it's not fully clear what, by default, should be the algorithm for the duplication.

For example, consider this object:

```
function anotherFunction() { /*..*/ }

var anotherObject = {
    c: true
};

var anotherArray = [];

var myObject = {
    a: 2,
    b: anotherObject,    // reference, not a copy!
    c: anotherArray,    // another reference!
    d: anotherFunction
};

anotherArray.push( anotherObject, myObject );
```

What exactly should be the representation of a *copy* of `myObject` ?

Firstly, we should answer if it should be a *shallow* or *deep* copy? A *shallow copy* would end up with `a` on the new object as a copy of the value `2`, but `b`, `c`, and `d` properties as just references to the same places as the references in the original object. A *deep copy* would duplicate not only `myObject`, but `anotherObject` and `anotherArray`. But then we have issues that `anotherArray` has references to `anotherObject` and `myObject` in it, so *those* should also be duplicated rather than reference-preserved. Now we have an infinite circular duplication problem because of the circular reference.

Should we detect a circular reference and just break the circular traversal (leaving the deep element not fully duplicated)? Should we error out completely? Something in between?

Moreover, it's not really clear what "duplicating" a function would mean? There are some hacks like pulling out the `toString()` serialization of a function's source code (which varies across implementations and is not even reliable in all engines depending on the type of function being inspected).

So how do we resolve all these tricky questions? Various JS frameworks have each picked their own interpretations and made their own decisions. But which of these (if any) should JS adopt as *the* standard? For a long time, there was no clear answer.

One subset solution is that objects which are JSON-safe (that is, can be serialized to a JSON string and then re-parsed to an object with the same structure and values) can easily be *duplicated* with:

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

Of course, that requires you to ensure your object is JSON safe. For some situations, that's trivial. For others, it's insufficient.

At the same time, a shallow copy is fairly understandable and has far less issues, so ES6 has now defined `Object.assign(..)` for this task. `Object.assign(..)` takes a *target* object as its first parameter, and one or more *source* objects as its subsequent parameters. It iterates over all the *enumerable* (see below), *owned keys* (**immediately present**) on the *source* object(s) and copies them (via `=` assignment only) to *target*. It also, helpfully, returns *target*, as you can see below:

```
var newObj = Object.assign( {}, myObject );

newObj.a;                        // 2
newObj.b === anotherObject;        // true
newObj.c === anotherArray;         // true
newObj.d === anotherFunction;    // true
```

Note: In the next section, we describe "property descriptors" (property characteristics) and show the use of `Object.defineProperty(..)`. The duplication that occurs for `Object.assign(..)` however is purely `=` style assignment, so any special characteristics of a property (like `writable`) on a source object **are not preserved** on the target object.

Property Descriptors

Prior to ES5, the JavaScript language gave no direct way for your code to inspect or draw any distinction between the characteristics of properties, such as whether the property was read-only or not.

But as of ES5, all properties are described in terms of a **property descriptor**.

Consider this code:

```
var myObject = {
    a: 2
};

Object.getOwnPropertyDescriptor( myObject, "a" );
// {
//    value: 2,
//    writable: true,
//    enumerable: true,
//    configurable: true
// }
```

As you can see, the property descriptor (called a "data descriptor" since it's only for holding a data value) for our normal object property `a` is much more than just its `value` of `2`. It includes 3 other characteristics: `writable`, `enumerable`, and `configurable`.

While we can see what the default values for the property descriptor characteristics are when we create a normal property, we can use `Object.defineProperty(..)` to add a new property, or modify an existing one (if it's `configurable`!), with the desired characteristics.

For example:

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: true,
    enumerable: true
} );

myObject.a; // 2
```

Using `defineProperty(..)`, we added the plain, normal `a` property to `myObject` in a manually explicit way. However, you generally wouldn't use this manual approach unless you wanted to modify one of the descriptor characteristics from its normal behavior.

Writable

The ability for you to change the value of a property is controlled by `writable`.

Consider:

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // not writable!
    configurable: true,
    enumerable: true
} );

myObject.a = 3;

myObject.a; // 2
```

As you can see, our modification of the `value` silently failed. If we try in `strict mode`, we get an error:

```
"use strict";

var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // not writable!
    configurable: true,
    enumerable: true
} );

myObject.a = 3; // TypeError
```

The `TypeError` tells us we cannot change a non-writable property.

Note: We will discuss getters/setters shortly, but briefly, you can observe that `writable:false` means a value cannot be changed, which is somewhat equivalent to if you defined a no-op setter. Actually, your no-op setter would need to throw a `TypeError` when called, to be truly conformant to `writable:false` .

Configurable

As long as a property is currently configurable, we can modify its descriptor definition, using the same `defineProperty(..)` utility.

```
var myObject = {
    a: 2
};

myObject.a = 3;
myObject.a;                 // 3

Object.defineProperty( myObject, "a", {
    value: 4,
    writable: true,
    configurable: false,    // not configurable!
    enumerable: true
} );

myObject.a;                 // 4
myObject.a = 5;
myObject.a;                 // 5

Object.defineProperty( myObject, "a", {
    value: 6,
    writable: true,
    configurable: true,
    enumerable: true
} ); // TypeError
```

The final `defineProperty(..)` call results in a TypeError, regardless of `strict mode`, if you attempt to change the descriptor definition of a non-configurable property. Be careful: as you can see, changing `configurable` to `false` is a **one-way action, and cannot be undone!**

Note: There's a nuanced exception to be aware of: even if the property is already `configurable:false`, `writable` can always be changed from `true` to `false` without error, but not back to `true` if already `false`.

Another thing `configurable:false` prevents is the ability to use the `delete` operator to remove an existing property.

```
var myObject = {
    a: 2
};

myObject.a;                 // 2
delete myObject.a;
myObject.a;                 // undefined

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: false,
    enumerable: true
} );

myObject.a;                 // 2
delete myObject.a;
myObject.a;                 // 2
```

As you can see, the last `delete` call failed (silently) because we made the `a` property non-configurable.

`delete` is only used to remove object properties (which can be removed) directly from the object in question. If an object property is the last remaining *reference* to some object/function, and you `delete` it, that removes the reference and now that unreferenced object/function can be garbage collected. But, it is **not** proper to think of `delete` as a tool to free up allocated memory as it does in other languages (like C/C++). `delete` is just an object property removal operation -- nothing more.

Enumerable

The final descriptor characteristic we will mention here (there are two others, which we deal with shortly when we discuss getter/setters) is `enumerable`.

The name probably makes it obvious, but this characteristic controls if a property will show up in certain object-property enumerations, such as the `for..in` loop. Set to `false` to keep it from showing up in such enumerations, even though it's still completely accessible. Set to `true` to keep it present.

All normal user-defined properties are defaulted to `enumerable`, as this is most commonly what you want. But if you have a special property you want to hide from enumeration, set it to `enumerable:false`.

We'll demonstrate enumerability in much more detail shortly, so keep a mental bookmark on this topic.

Immutability

It is sometimes desired to make properties or objects that cannot be changed (either by accident or intentionally). ES5 adds support for handling that in a variety of different nuanced ways.

It's important to note that **all** of these approaches create shallow immutability. That is, they affect only the object and its direct property characteristics. If an object has a reference to another object (array, object, function, etc), the *contents* of that object are not affected, and remain mutable.

```
myImmutableObject.foo; // [1,2,3]
myImmutableObject.foo.push( 4 );
myImmutableObject.foo; // [1,2,3,4]
```

We assume in this snippet that `myImmutableObject` is already created and protected as immutable. But, to also protect the contents of `myImmutableObject.foo` (which is its own object -- array), you would also need to make `foo` immutable, using one or more of the following functionalities.

Note: It is not terribly common to create deeply entrenched immutable objects in JS programs. Special cases can certainly call for it, but as a general design pattern, if you find yourself wanting to *seal* or *freeze* all your objects, you may want to take a step back and reconsider your program design to be more robust to potential changes in objects' values.

Object Constant

By combining `writable:false` and `configurable:false`, you can essentially create a *constant* (cannot be changed, redefined or deleted) as an object property, like:

```
var myObject = {};

Object.defineProperty( myObject, "FAVORITE_NUMBER", {
    value: 42,
    writable: false,
    configurable: false
} );
```

Prevent Extensions

If you want to prevent an object from having new properties added to it, but otherwise leave the rest of the object's properties alone, call `Object.preventExtensions(..)`:

```
var myObject = {
    a: 2
};

Object.preventExtensions( myObject );

myObject.b = 3;
myObject.b; // undefined
```

In `non-strict mode`, the creation of `b` fails silently. In `strict mode`, it throws a `TypeError`.

Seal

`Object.seal(..)` creates a "sealed" object, which means it takes an existing object and essentially calls `Object.preventExtensions(..)` on it, but also marks all its existing properties as `configurable:false`.

So, not only can you not add any more properties, but you also cannot reconfigure or delete any existing properties (though you *can* still modify their values).

Freeze

`Object.freeze(..)` creates a frozen object, which means it takes an existing object and essentially calls `Object.seal(..)` on it, but it also marks all "data accessor" properties as `writable:false`, so that their values cannot be changed.

This approach is the highest level of immutability that you can attain for an object itself, as it prevents any changes to the object or to any of its direct properties (though, as mentioned above, the contents of any referenced other objects are unaffected).

You could "deep freeze" an object by calling `Object.freeze(..)` on the object, and then recursively iterating over all objects it references (which would have been unaffected thus far), and calling `Object.freeze(..)` on them as well. Be careful, though, as that could affect other (shared) objects you're not intending to affect.

`[[Get]]`

There's a subtle, but important, detail about how property accesses are performed.

Consider:

```
var myObject = {
    a: 2
};

myObject.a; // 2
```

The `myObject.a` is a property access, but it doesn't *just* look in `myObject` for a property of the name `a`, as it might seem.

According to the spec, the code above actually performs a `[[Get]]` operation (kinda like a function call: `[[Get]]()`) on the `myObject`. The default built-in `[[Get]]` operation for an object *first* inspects the object for a property of the requested name, and if it finds it, it will return the value accordingly.

However, the `[[Get]]` algorithm defines other important behavior if it does *not* find a property of the requested name. We will examine in Chapter 5 what happens *next* (traversal of the `[[Prototype]]` chain, if any).

But one important result of this `[[Get]]` operation is that if it cannot through any means come up with a value for the requested property, it instead returns the value `undefined`.

```
var myObject = {
    a: 2
};

myObject.b; // undefined
```

This behavior is different from when you reference *variables* by their identifier names. If you reference a variable that cannot be resolved within the applicable lexical scope look-up, the result is not `undefined` as it is for object properties, but instead a `ReferenceError` is thrown.

```
var myObject = {
    a: undefined
};

myObject.a; // undefined

myObject.b; // undefined
```

From a *value* perspective, there is no difference between these two references -- they both result in `undefined`. However, the `[[Get]]` operation underneath, though subtle at a glance, potentially performed a bit more "work" for the reference `myObject.b` than for the reference `myObject.a`.

Inspecting only the value results, you cannot distinguish whether a property exists and holds the explicit value `undefined`, or whether the property does *not* exist and `undefined` was the default return value after `[[Get]]` failed to return something explicitly. However, we will see shortly how you *can* distinguish these two scenarios.

`[[Put]]`

Since there's an internally defined `[[Get]]` operation for getting a value from a property, it should be obvious there's also a default `[[Put]]` operation.

It may be tempting to think that an assignment to a property on an object would just invoke `[[Put]]` to set or create that property on the object in question. But the situation is more nuanced than that.

When invoking `[[Put]]`, how it behaves differs based on a number of factors, including (most impactfully) whether the property is already present on the object or not.

If the property is present, the `[[Put]]` algorithm will roughly check:

1. Is the property an accessor descriptor (see "Getters & Setters" section below)? **If so, call the setter, if any.**
2. Is the property a data descriptor with `writable` of `false`? **If so, silently fail in `non-strict mode`, or throw `TypeError` in `strict mode`.**
3. Otherwise, set the value to the existing property as normal.

If the property is not yet present on the object in question, the `[[Put]]` operation is even more nuanced and complex. We will revisit this scenario in Chapter 5 when we discuss `[[Prototype]]` to give it more clarity.

Getters & Setters

The default `[[Put]]` and `[[Get]]` operations for objects completely control how values are set to existing or new properties, or retrieved from existing properties, respectively.

Note: Using future/advanced capabilities of the language, it may be possible to override the default `[[Get]]` or `[[Put]]` operations for an entire object (not just per property). This is beyond the scope of our discussion in this book, but will be covered later in the "You Don't Know JS" series.

ES5 introduced a way to override part of these default operations, not on an object level but a per-property level, through the use of getters and setters. Getters are properties which actually call a hidden function to retrieve a value. Setters are properties which actually call a hidden function to set a value.

When you define a property to have either a getter or a setter or both, its definition becomes an "accessor descriptor" (as opposed to a "data descriptor"). For accessor-descriptors, the `value` and `writable` characteristics of the descriptor are moot and ignored, and instead JS considers the `set` and `get` characteristics of the property (as well as `configurable` and `enumerable`).

Consider:

```
var myObject = {
    // define a getter for `a`
    get a() {
        return 2;
    }
};

Object.defineProperty(
    myObject,    // target
    "b",        // property name
    {            // descriptor
        // define a getter for `b`
        get: function(){ return this.a * 2 },

        // make sure `b` shows up as an object property
        enumerable: true
    }
);

myObject.a; // 2

myObject.b; // 4
```

Either through object-literal syntax with `get a() { .. }` or through explicit definition with `defineProperty(..)` , in both cases we created a property on the object that actually doesn't hold a value, but whose access automatically results in a hidden function call to the getter function, with whatever value it returns being the result of the property access.

```
var myObject = {
    // define a getter for `a`
    get a() {
        return 2;
    }
};

myObject.a = 3;

myObject.a; // 2
```

Since we only defined a getter for `a`, if we try to set the value of `a` later, the set operation won't throw an error but will just silently throw the assignment away. Even if there was a valid setter, our custom getter is hard-coded to return only `2`, so the set operation would be moot.

To make this scenario more sensible, properties should also be defined with setters, which override the default `[[Put]]` operation (aka, assignment), per-property, just as you'd expect. You will almost certainly want to always declare both getter and setter (having only one or the other often leads to unexpected/surprising behavior):

```
var myObject = {
    // define a getter for `a`
    get a() {
        return this._a_;
    },

    // define a setter for `a`
    set a(val) {
        this._a_ = val * 2;
    }
};

myObject.a = 2;

myObject.a; // 4
```

Note: In this example, we actually store the specified value `2` of the assignment (`[[Put]]` operation) into another variable `_a_`. The `_a_` name is purely by convention for this example and implies nothing special about its behavior -- it's a normal property like any other.

Existence

We showed earlier that a property access like `myObject.a` may result in an `undefined` value if either the explicit `undefined` is stored there or the `a` property doesn't exist at all. So, if the value is the same in both cases, how else do we distinguish them?

We can ask an object if it has a certain property *without* asking to get that property's value:

```
var myObject = {
    a: 2
};

("a" in myObject);                // true
("b" in myObject);                // false

myObject.hasOwnProperty( "a" );    // true
myObject.hasOwnProperty( "b" );    // false
```

The `in` operator will check to see if the property is *in* the object, or if it exists at any higher level of the `[[Prototype]]` chain object traversal (see Chapter 5). By contrast, `hasOwnProperty(..)` checks to see if *only* `myObject` has the property or not, and will *not* consult the `[[Prototype]]` chain. We'll come back to the important differences between these two operations in Chapter 5 when we explore `[[Prototype]]`s in detail.

`hasOwnProperty(..)` is accessible for all normal objects via delegation to `Object.prototype` (see Chapter 5). But it's possible to create an object that does not link to `Object.prototype` (via `Object.create(null)` -- see Chapter 5). In this case, a method call like `myObject.hasOwnProperty(..)` would fail.

In that scenario, a more robust way of performing such a check is `Object.prototype.hasOwnProperty.call(myObject,"a")`, which borrows the base `hasOwnProperty(..)` method and uses *explicit* `this` *binding* (see Chapter 2) to apply it against our `myObject`.

Note: The `in` operator has the appearance that it will check for the existence of a *value* inside a container, but it actually checks for the existence of a property name. This difference is important to note with respect to arrays, as the temptation to try a check like `4 in [2, 4, 6]` is strong, but this will not behave as expected.

Enumeration

Previously, we explained briefly the idea of "enumerability" when we looked at the `enumerable` property descriptor characteristic. Let's revisit that and examine it in more close detail.

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // make `a` enumerable, as normal
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // make `b` NON-enumerable
    { enumerable: false, value: 3 }
);

myObject.b; // 3
("b" in myObject); // true
myObject.hasOwnProperty( "b" ); // true

// .......

for (var k in myObject) {
    console.log( k, myObject[k] );
}
// "a" 2
```

You'll notice that `myObject.b` in fact **exists** and has an accessible value, but it doesn't show up in a `for..in` loop (though, surprisingly, it **is** revealed by the `in` operator existence check). That's because "enumerable" basically means "will be included if the object's properties are iterated through".

Note: `for..in` loops applied to arrays can give somewhat unexpected results, in that the enumeration of an array will include not only all the numeric indices, but also any enumerable properties. It's a good idea to use `for..in` loops *only* on objects, and traditional `for` loops with numeric index iteration for the values stored in arrays.

Another way that enumerable and non-enumerable properties can be distinguished:

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // make `a` enumerable, as normal
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // make `b` non-enumerable
    { enumerable: false, value: 3 }
);

myObject.propertyIsEnumerable( "a" ); // true
myObject.propertyIsEnumerable( "b" ); // false

Object.keys( myObject ); // ["a"]
Object.getOwnPropertyNames( myObject ); // ["a", "b"]
```

`propertyIsEnumerable(..)` tests whether the given property name exists *directly* on the object and is also `enumerable:true` .

`Object.keys(..)` returns an array of all enumerable properties, whereas `Object.getOwnPropertyNames(..)` returns an array of *all* properties, enumerable or not.

Whereas `in` vs. `hasOwnProperty(..)` differ in whether they consult the `[[Prototype]]` chain or not, `Object.keys(..)` and `Object.getOwnPropertyNames(..)` both inspect *only* the direct object specified.

There's (currently) no built-in way to get a list of **all properties** which is equivalent to what the `in` operator test would consult (traversing all properties on the entire `[[Prototype]]` chain, as explained in Chapter 5). You could approximate such a utility by recursively traversing the `[[Prototype]]` chain of an object, and for each level, capturing the list from `Object.keys(..)` -- only enumerable properties.

Iteration

The `for..in` loop iterates over the list of enumerable properties on an object (including its `[[Prototype]]` chain). But what if you instead want to iterate over the values?

With numerically-indexed arrays, iterating over the values is typically done with a standard `for` loop, like:

```
var myArray = [1, 2, 3];

for (var i = 0; i < myArray.length; i++) {
    console.log( myArray[i] );
}
// 1 2 3
```

This isn't iterating over the values, though, but iterating over the indices, where you then use the index to reference the value, as `myArray[i]`.

ES5 also added several iteration helpers for arrays, including `forEach(..)`, `every(..)`, and `some(..)`. Each of these helpers accepts a function callback to apply to each element in the array, differing only in they respectively respond to a return value from the callback.

`forEach(..)` will iterate over all values in the array, and ignores any callback return values. `every(..)` keeps going until the end *or* the callback returns a `false` (or "falsy") value, whereas `some(..)` keeps going until the end *or* the callback returns a `true` (or "truthy") value.

These special return values inside `every(..)` and `some(..)` act somewhat like a `break` statement inside a normal `for` loop, in that they stop the iteration early before it reaches the end.

If you iterate on an object with a `for..in` loop, you're also only getting at the values indirectly, because it's actually iterating only over the enumerable properties of the object, leaving you to access the properties manually to get the values.

Note: As contrasted with iterating over an array's indices in a numerically ordered way (`for` loop or other iterators), the order of iteration over an object's properties is **not guaranteed** and may vary between different JS engines. **Do not rely** on any observed ordering for anything that requires consistency among environments, as any observed agreement is unreliable.

But what if you want to iterate over the values directly instead of the array indices (or object properties)? Helpfully, ES6 adds a `for..of` loop syntax for iterating over arrays (and objects, if the object defines its own custom iterator):

```
var myArray = [ 1, 2, 3 ];

for (var v of myArray) {
    console.log( v );
}
// 1
// 2
// 3
```

The `for..of` loop asks for an iterator object (from a default internal function known as `@@iterator` in spec-speak) of the *thing* to be iterated, and the loop then iterates over the successive return values from calling that iterator object's `next()` method, once for each loop iteration.

Arrays have a built-in `@@iterator`, so `for..of` works easily on them, as shown. But let's manually iterate the array, using the built-in `@@iterator`, to see how it works:

```
var myArray = [ 1, 2, 3 ];
var it = myArray[Symbol.iterator]();

it.next(); // { value:1, done:false }
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { done:true }
```

Note: We get at the `@@iterator` *internal property* of an object using an ES6 `Symbol`: `Symbol.iterator`. We briefly mentioned `Symbol` semantics earlier in the chapter (see "Computed Property Names"), so the same reasoning applies here. You'll always want to reference such special properties by `Symbol` name reference instead of by the special value it may hold. Also, despite the name's implications, `@@iterator` is **not the iterator object** itself, but a **function that returns** the iterator object -- a subtle but important detail!

As the above snippet reveals, the return value from an iterator's `next()` call is an object of the form `{ value: .. , done: .. }`, where `value` is the current iteration value, and `done` is a `boolean` that indicates if there's more to iterate.

Notice the value `3` was returned with a `done:false`, which seems strange at first glance. You have to call the `next()` a fourth time (which the `for..of` loop in the previous snippet automatically does) to get `done:true` and know you're truly done iterating. The reason for this quirk is beyond the scope of what we'll discuss here, but it comes from the semantics of ES6 generator functions.

While arrays do automatically iterate in `for..of` loops, regular objects **do not have a built-in `@@iterator`**. The reasons for this intentional omission are more complex than we will examine here, but in general it was better to not include some implementation that could prove troublesome for future types of objects.

It *is* possible to define your own default `@@iterator` for any object that you care to iterate over. For example:

```
var myObject = {
    a: 2,
    b: 3
};

Object.defineProperty( myObject, Symbol.iterator, {
    enumerable: false,
    writable: false,
    configurable: true,
    value: function() {
        var o = this;
        var idx = 0;
        var ks = Object.keys( o );
        return {
            next: function() {
                return {
                    value: o[ks[idx++]],
                    done: (idx > ks.length)
                };
            }
        };
    }
} );

// iterate `myObject` manually
var it = myObject[Symbol.iterator]();
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { value:undefined, done:true }

// iterate `myObject` with `for..of`
for (var v of myObject) {
    console.log( v );
}
// 2
// 3
```

Note: We used `Object.defineProperty(..)` to define our custom `@@iterator` (mostly so we could make it non-enumerable), but using the `Symbol` as a *computed property name* (covered earlier in this chapter), we could have declared it directly, like `var myObject = { a:2, b:3, [Symbol.iterator]: function(){ /* .. */ } }`.

Each time the `for..of` loop calls `next()` on `myObject`'s iterator object, the internal pointer will advance and return back the next value from the object's properties list (see a previous note about iteration ordering on object properties/values).

The iteration we just demonstrated is a simple value-by-value iteration, but you can of course define arbitrarily complex iterations for your custom data structures, as you see fit. Custom iterators combined with ES6's `for..of` loop are a powerful new syntactic tool for

manipulating user-defined objects.

For example, a list of `Pixel` objects (with `x` and `y` coordinate values) could decide to order its iteration based on the linear distance from the `(0,0)` origin, or filter out points that are "too far away", etc. As long as your iterator returns the expected `{ value: .. }` return values from `next()` calls, and a `{ done: true }` after the iteration is complete, ES6's `for..of` can iterate over it.

In fact, you can even generate "infinite" iterators which never "finish" and always return a new value (such as a random number, an incremented value, a unique identifier, etc), though you probably will not use such iterators with an unbounded `for..of` loop, as it would never end and would hang your program.

```
var randoms = {
    [Symbol.iterator]: function() {
        return {
            next: function() {
                return { value: Math.random() };
            }
        };
    }
};

var randoms_pool = [];
for (var n of randoms) {
    randoms_pool.push( n );

    // don't proceed unbounded!
    if (randoms_pool.length === 100) break;
}
```

This iterator will generate random numbers "forever", so we're careful to only pull out 100 values so our program doesn't hang.

Review (TL;DR)

Objects in JS have both a literal form (such as `var a = { .. }`) and a constructed form (such as `var a = new Array(..)`). The literal form is almost always preferred, but the constructed form offers, in some cases, more creation options.

Many people mistakingly claim "everything in JavaScript is an object", but this is incorrect. Objects are one of the 6 (or 7, depending on your perspective) primitive types. Objects have sub-types, including `function`, and also can be behavior-specialized, like `[object Array]` as the internal label representing the array object sub-type.

Objects are collections of key/value pairs. The values can be accessed as properties, via `.propName` or `["propName"]` syntax. Whenever a property is accessed, the engine actually invokes the internal default `[[Get]]` operation (and `[[Put]]` for setting values), which not only looks for the property directly on the object, but which will traverse the `[[Prototype]]` chain (see Chapter 5) if not found.

Properties have certain characteristics that can be controlled through property descriptors, such as `writable` and `configurable`. In addition, objects can have their mutability (and that of their properties) controlled to various levels of immutability using `Object.preventExtensions(..)`, `Object.seal(..)`, and `Object.freeze(..)`.

Properties don't have to contain values -- they can be "accessor properties" as well, with getters/setters. They can also be either *enumerable* or not, which controls if they show up in `for..in` loop iterations, for instance.

You can also iterate over **the values** in data structures (arrays, objects, etc) using the ES6 `for..of` syntax, which looks for either a built-in or custom `@@iterator` object consisting of a `next()` method to advance through the data values one at a time.

You Don't Know JS: *this* & Object Prototypes

Chapter 4: Mixing (Up) "Class" Objects

Following our exploration of objects from the previous chapter, it's natural that we now turn our attention to "object oriented (OO) programming", with "classes". We'll first look at "class orientation" as a design pattern, before examining the mechanics of "classes": "instantiation", "inheritance" and "(relative) polymorphism".

We'll see that these concepts don't really map very naturally to the object mechanism in JS, and the lengths (mixins, etc.) many JavaScript developers go to overcome such challenges.

Note: This chapter spends quite a bit of time (the first half!) on heavy "objected oriented programming" theory. We eventually relate these ideas to real concrete JavaScript code in the second half, when we talk about "Mixins". But there's a lot of concept and pseudo-code to wade through first, so don't get lost -- just stick with it!

Class Theory

"Class/Inheritance" describes a certain form of code organization and architecture -- a way of modeling real world problem domains in our software.

OO or class oriented programming stresses that data intrinsically has associated behavior (of course, different depending on the type and nature of the data!) that operates on it, so proper design is to package up (aka, encapsulate) the data and the behavior together. This is sometimes called "data structures" in formal computer science.

For example, a series of characters that represents a word or phrase is usually called a "string". The characters are the data. But you almost never just care about the data, you usually want to *do things* with the data, so the behaviors that can apply *to* that data (calculating its length, appending data, searching, etc.) are all designed as methods of a `String` class.

Any given string is just an instance of this class, which means that it's a neatly collected packaging of both the character data and the functionality we can perform on it.

Classes also imply a way of *classifying* a certain data structure. The way we do this is to think about any given structure as a specific variation of a more general base definition.

Let's explore this classification process by looking at a commonly cited example. A *car* can be described as a specific implementation of a more general "class" of thing, called a *vehicle*.

We model this relationship in software with classes by defining a `Vehicle` class and a `Car` class.

The definition of `Vehicle` might include things like propulsion (engines, etc.), the ability to carry people, etc., which would all be the behaviors. What we define in `Vehicle` is all the stuff that is common to all (or most of) the different types of vehicles (the "planes, trains, and automobiles").

It might not make sense in our software to re-define the basic essence of "ability to carry people" over and over again for each different type of vehicle. Instead, we define that capability once in `Vehicle`, and then when we define `Car`, we simply indicate that it "inherits" (or "extends") the base definition from `Vehicle`. The definition of `Car` is said to specialize the general `Vehicle` definition.

While `Vehicle` and `Car` collectively define the behavior by way of methods, the data in an instance would be things like the unique VIN of a specific car, etc.

And thus, classes, inheritance, and instantiation emerge.

Another key concept with classes is "polymorphism", which describes the idea that a general behavior from a parent class can be overridden in a child class to give it more specifics. In fact, relative polymorphism lets us reference the base behavior from the overridden behavior.

Class theory strongly suggests that a parent class and a child class share the same method name for a certain behavior, so that the child overrides the parent (differentially). As we'll see later, doing so in your JavaScript code is opting into frustration and code brittleness.

"Class" Design Pattern

You may never have thought about classes as a "design pattern", since it's most common to see discussion of popular "OO Design Patterns", like "Iterator", "Observer", "Factory", "Singleton", etc. As presented this way, it's almost an assumption that OO classes are the lower-level mechanics by which we implement all (higher level) design patterns, as if OO is a given foundation for *all* (proper) code.

Depending on your level of formal education in programming, you may have heard of "procedural programming" as a way of describing code which only consists of procedures (aka, functions) calling other functions, without any higher abstractions. You may have been

taught that classes were the *proper* way to transform procedural-style "spaghetti code" into well-formed, well-organized code.

Of course, if you have experience with "functional programming" (Monads, etc.), you know very well that classes are just one of several common design patterns. But for others, this may be the first time you've asked yourself if classes really are a fundamental foundation for code, or if they are an optional abstraction on top of code.

Some languages (like Java) don't give you the choice, so it's not very *optional* at all -- everything's a class. Other languages like C/C++ or PHP give you both procedural and class-oriented syntaxes, and it's left more to the developer's choice which style or mixture of styles is appropriate.

JavaScript "Classes"

Where does JavaScript fall in this regard? JS has had *some* class-like syntactic elements (like `new` and `instanceof`) for quite awhile, and more recently in ES6, some additions, like the `class` keyword (see Appendix A).

But does that mean JavaScript actually *has* classes? Plain and simple: **No.**

Since classes are a design pattern, you *can*, with quite a bit of effort (as we'll see throughout the rest of this chapter), implement approximations for much of classical class functionality. JS tries to satisfy the extremely pervasive *desire* to design with classes by providing seemingly class-like syntax.

While we may have a syntax that looks like classes, it's as if JavaScript mechanics are fighting against you using the *class design pattern*, because behind the curtain, the mechanisms that you build on are operating quite differently. Syntactic sugar and (extremely widely used) JS "Class" libraries go a long way toward hiding this reality from you, but sooner or later you will face the fact that the *classes* you have in other languages are not like the "classes" you're faking in JS.

What this boils down to is that classes are an optional pattern in software design, and you have the choice to use them in JavaScript or not. Since many developers have a strong affinity to class oriented software design, we'll spend the rest of this chapter exploring what it takes to maintain the illusion of classes with what JS provides, and the pain points we experience.

Class Mechanics

In many class-oriented languages, the "standard library" provides a "stack" data structure (push, pop, etc.) as a `Stack` class. This class would have an internal set of variables that stores the data, and it would have a set of publicly accessible behaviors ("methods") provided by the class, which gives your code the ability to interact with the (hidden) data (adding & removing data, etc.).

But in such languages, you don't really operate directly on `Stack` (unless making a **Static** class member reference, which is outside the scope of our discussion). The `Stack` class is merely an abstract explanation of what *any* "stack" should do, but it's not itself *a* "stack". You must **instantiate** the `Stack` class before you have a concrete data structure *thing* to operate against.

Building

The traditional metaphor for "class" and "instance" based thinking comes from a building construction.

An architect plans out all the characteristics of a building: how wide, how tall, how many windows and in what locations, even what type of material to use for the walls and roof. She doesn't necessarily care, at this point, *where* the building will be built, nor does she care *how many* copies of that building will be built.

She also doesn't care very much about the contents of the building -- the furniture, wall paper, ceiling fans, etc. -- only what type of structure they will be contained by.

The architectural blue-prints she produces are only *plans* for a building. They don't actually constitute a building we can walk into and sit down. We need a builder for that task. A builder will take those plans and follow them, exactly, as he *builds* the building. In a very real sense, he is *copying* the intended characteristics from the plans to the physical building.

Once complete, the building is a physical instantiation of the blue-print plans, hopefully an essentially perfect *copy*. And then the builder can move to the open lot next door and do it all over again, creating yet another *copy*.

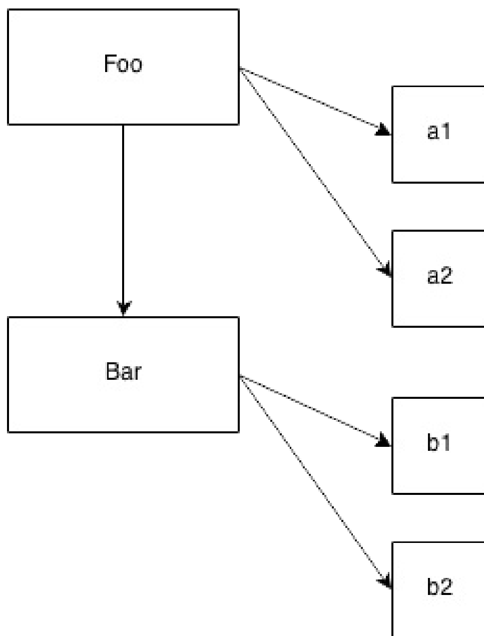
The relationship between building and blue-print is indirect. You can examine a blue-print to understand how the building was structured, for any parts where direct inspection of the building itself was insufficient. But if you want to open a door, you have to go to the building itself -- the blue-print merely has lines drawn on a page that *represent* where the door should be.

A class is a blue-print. To actually *get* an object we can interact with, we must build (aka, "instantiate") something from the class. The end result of such "construction" is an object, typically called an "instance", which we can directly call methods on and access any public data properties from, as necessary.

This object is a *copy* of all the characteristics described by the class.

You likely wouldn't expect to walk into a building and find, framed and hanging on the wall, a copy of the blue-prints used to plan the building, though the blue-prints are probably on file with a public records office. Similarly, you don't generally use an object instance to directly access and manipulate its class, but it is usually possible to at least determine *which class* an object instance comes from.

It's more useful to consider the direct relationship of a class to an object instance, rather than any indirect relationship between an object instance and the class it came from. **A class is instantiated into object form by a copy operation.**



As you can see, the arrows move from left to right, and from top to bottom, which indicates the copy operations that occur, both conceptually and physically.

Constructor

Instances of classes are constructed by a special method of the class, usually of the same name as the class, called a *constructor*. This method's explicit job is to initialize any information (state) the instance will need.

For example, consider this loose pseudo-code (invented syntax) for classes:

```
class CoolGuy {
    specialTrick = nothing

    CoolGuy( trick ) {
        specialTrick = trick
    }

    showOff() {
        output( "Here's my trick: ", specialTrick )
    }
}
```

To *make* a `CoolGuy` instance, we would call the class constructor:

```
Joe = new CoolGuy( "jumping rope" )

Joe.showOff() // Here's my trick: jumping rope
```

Notice that the `CoolGuy` class has a constructor `CoolGuy()`, which is actually what we call when we say `new CoolGuy(..)`. We get an object back (an instance of our class) from the constructor, and we can call the method `showOff()`, which prints out that particular `CoolGuy`s special trick.

Obviously, jumping rope makes Joe a pretty cool guy.

The constructor of a class *belongs* to the class, almost universally with the same name as the class. Also, constructors pretty much always need to be called with `new` to let the language engine know you want to construct a *new* class instance.

Class Inheritance

In class-oriented languages, not only can you define a class which can be instantiated itself, but you can define another class that **inherits** from the first class.

The second class is often said to be a "child class" whereas the first is the "parent class". These terms obviously come from the metaphor of parents and children, though the metaphors here are a bit stretched, as you'll see shortly.

When a parent has a biological child, the genetic characteristics of the parent are copied into the child. Obviously, in most biological reproduction systems, there are two parents who co-equally contribute genes to the mix. But for the purposes of the metaphor, we'll assume just one parent.

Once the child exists, he or she is separate from the parent. The child was heavily influenced by the inheritance from his or her parent, but is unique and distinct. If a child ends up with red hair, that doesn't mean the parent's hair *was* or automatically *becomes* red.

In a similar way, once a child class is defined, it's separate and distinct from the parent class. The child class contains an initial copy of the behavior from the parent, but can then override any inherited behavior and even define new behavior.

It's important to remember that we're talking about parent and child **classes**, which aren't physical things. This is where the metaphor of parent and child gets a little confusing, because we actually should say that a parent class is like a parent's DNA and a child class is like a child's DNA. We have to make (aka "instantiate") a person out of each set of DNA to actually have a physical person to have a conversation with.

Let's set aside biological parents and children, and look at inheritance through a slightly different lens: different types of vehicles. That's one of the most canonical (and often groan-worthy) metaphors to understand inheritance.

Let's revisit the `Vehicle` and `Car` discussion from earlier in this chapter. Consider this loose pseudo-code (invented syntax) for inherited classes:

```
class Vehicle {
    engines = 1

    ignition() {
        output( "Turning on my engine." )
    }

    drive() {
        ignition()
        output( "Steering and moving forward!" )
    }
}

class Car inherits Vehicle {
    wheels = 4

    drive() {
        inherited:drive()
        output( "Rolling on all ", wheels, " wheels!" )
    }
}

class SpeedBoat inherits Vehicle {
    engines = 2

    ignition() {
        output( "Turning on my ", engines, " engines." )
    }

    pilot() {
        inherited:drive()
        output( "Speeding through the water with ease!" )
    }
}
```

Note: For clarity and brevity, constructors for these classes have been omitted.

We define the `Vehicle` class to assume an engine, a way to turn on the ignition, and a way to drive around. But you wouldn't ever manufacture just a generic "vehicle", so it's really just an abstract concept at this point.

So then we define two specific kinds of vehicle: `Car` and `SpeedBoat`. They each inherit the general characteristics of `Vehicle`, but then they specialize the characteristics appropriately for each kind. A car needs 4 wheels, and a speed boat needs 2 engines, which means it needs extra attention to turn on the ignition of both engines.

Polymorphism

`Car` defines its own `drive()` method, which overrides the method of the same name it inherited from `Vehicle`. But then, `Car` s `drive()` method calls `inherited:drive()`, which indicates that `Car` can reference the original pre-overridden `drive()` it inherited. `SpeedBoat` s `pilot()` method also makes a reference to its inherited copy of `drive()`.

This technique is called "polymorphism", or "virtual polymorphism". More specifically to our current point, we'll call it "relative polymorphism".

Polymorphism is a much broader topic than we will exhaust here, but our current "relative" semantics refers to one particular aspect: the idea that any method can reference another method (of the same or different name) at a higher level of the inheritance hierarchy. We say "relative" because we don't absolutely define which inheritance level (aka, class) we want to access, but rather relatively reference it by essentially saying "look one level up".

In many languages, the keyword `super` is used, in place of this example's `inherited:`, which leans on the idea that a "super class" is the parent/ancestor of the current class.

Another aspect of polymorphism is that a method name can have multiple definitions at different levels of the inheritance chain, and these definitions are automatically selected as appropriate when resolving which methods are being called.

We see two occurrences of that behavior in our example above: `drive()` is defined in both `Vehicle` and `Car`, and `ignition()` is defined in both `Vehicle` and `SpeedBoat`.

Note: Another thing that traditional class-oriented languages give you via `super` is a direct way for the constructor of a child class to reference the constructor of its parent class. This is largely true because with real classes, the constructor belongs to the class. However, in JS, it's the reverse -- it's actually more appropriate to think of the "class" belonging to the constructor (the `Foo.prototype...` type references). Since in JS the relationship between child and parent exists only between the two `.prototype` objects of the respective constructors, the constructors themselves are not directly related, and thus there's no simple way to relatively reference one from the other (see Appendix A for ES6 `class` which "solves" this with `super`).

An interesting implication of polymorphism can be seen specifically with `ignition()`. Inside `pilot()`, a relative-polymorphic reference is made to (the inherited) `Vehicle`s version of `drive()`. But that `drive()` references an `ignition()` method just by name (no relative reference).

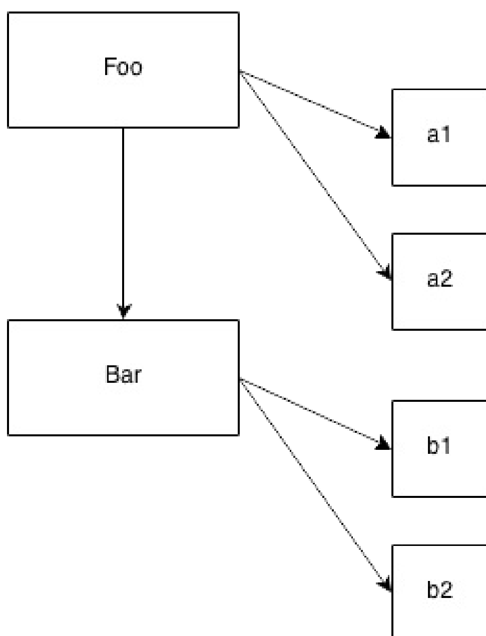
Which version of `ignition()` will the language engine use, the one from `Vehicle` or the one from `SpeedBoat`? **It uses the `SpeedBoat` version of `ignition()`.** If you *were* to instantiate `Vehicle` class itself, and then call its `drive()`, the language engine would instead just use `Vehicle` s `ignition()` method definition.

Put another way, the definition for the method `ignition()` *polymorphs* (changes) depending on which class (level of inheritance) you are referencing an instance of.

This may seem like overly deep academic detail. But understanding these details is necessary to properly contrast similar (but distinct) behaviors in JavaScript's `[[Prototype]]` mechanism.

When classes are inherited, there is a way **for the classes themselves** (not the object instances created from them!) to *relatively* reference the class inherited from, and this relative reference is usually called `super` .

Remember this figure from earlier:



Notice how for both instantiation (`a1` , `a2` , `b1` , and `b2`) *and* inheritance (`Bar`), the arrows indicate a copy operation.

Conceptually, it would seem a child class `Bar` can access behavior in its parent class `Foo` using a relative polymorphic reference (aka, `super`). However, in reality, the child class is merely given a copy of the inherited behavior from its parent class. If the child "overrides" a method it inherits, both the original and overridden versions of the method are actually maintained, so that they are both accessible.

Don't let polymorphism confuse you into thinking a child class is linked to its parent class. A child class instead gets a copy of what it needs from the parent class. **Class inheritance implies copies.**

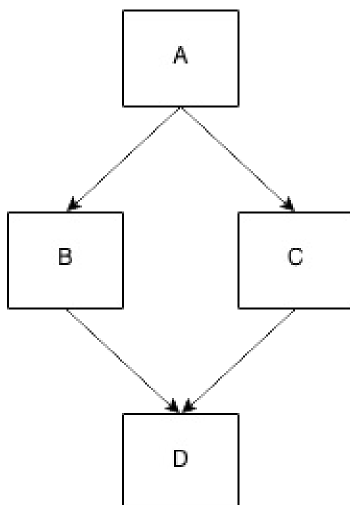
Multiple Inheritance

Recall our earlier discussion of parent(s) and children and DNA? We said that the metaphor was a bit weird because biologically most offspring come from two parents. If a class could inherit from two other classes, it would more closely fit the parent/child metaphor.

Some class-oriented languages allow you to specify more than one "parent" class to "inherit" from. Multiple-inheritance means that each parent class definition is copied into the child class.

On the surface, this seems like a powerful addition to class-orientation, giving us the ability to compose more functionality together. However, there are certainly some complicating questions that arise. If both parent classes provide a method called `drive()`, which version would a `drive()` reference in the child resolve to? Would you always have to manually specify which parent's `drive()` you meant, thus losing some of the gracefulness of polymorphic inheritance?

There's another variation, the so called "Diamond Problem", which refers to the scenario where a child class "D" inherits from two parent classes ("B" and "C"), and each of those in turn inherits from a common "A" parent. If "A" provides a method `drive()`, and both "B" and "C" override (polymorph) that method, when `D` references `drive()`, which version should it use (`B:drive()` or `C:drive()`)?



These complications go even much deeper than this quick glance. We address them here only so we can contrast to how JavaScript's mechanisms work.

JavaScript is simpler: it does not provide a native mechanism for "multiple inheritance". Many see this is a good thing, because the complexity savings more than make up for the "reduced" functionality. But this doesn't stop developers from trying to fake it in various ways, as we'll see next.

Mixins

JavaScript's object mechanism does not *automatically* perform copy behavior when you "inherit" or "instantiate". Plainly, there are no "classes" in JavaScript to instantiate, only objects. And objects don't get copied to other objects, they get *linked together* (more on that in Chapter 5).

Since observed class behaviors in other languages imply copies, let's examine how JS developers **fake** the *missing* copy behavior of classes in JavaScript: mixins. We'll look at two types of "mixin": **explicit** and **implicit**.

Explicit Mixins

Let's again revisit our `Vehicle` and `Car` example from before. Since JavaScript will not automatically copy behavior from `Vehicle` to `Car`, we can instead create a utility that manually copies. Such a utility is often called `extend(..)` by many libraries/frameworks, but we will call it `mixin(..)` here for illustrative purposes.

```
// vastly simplified `mixin(..)` example:
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // only copy if not already present
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}

var Vehicle = {
    engines: 1,

    ignition: function() {
        console.log( "Turning on my engine." );
    },

    drive: function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
    }
};

var Car = mixin( Vehicle, {
    wheels: 4,

    drive: function() {
        Vehicle.drive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    }
} );
```

Note: Subtly but importantly, we're not dealing with classes anymore, because there are no classes in JavaScript. `Vehicle` and `Car` are just objects that we make copies from and to, respectively.

`Car` now has a copy of the properties and functions from `Vehicle`. Technically, functions are not actually duplicated, but rather *references* to the functions are copied. So, `Car` now has a property called `ignition`, which is a copied reference to the `ignition()` function, as well as a property called `engines` with the copied value of `1` from `Vehicle`.

`Car` *already* had a `drive` property (function), so that property reference was not overridden (see the `if` statement in `mixin(..)` above).

"Polymorphism" Revisited

Let's examine this statement: `Vehicle.drive.call( this )`. This is what I call "explicit pseudo-polymorphism". Recall in our previous pseudo-code this line was `inherited:drive()`, which we called "relative polymorphism".

JavaScript does not have (prior to ES6; see Appendix A) a facility for relative polymorphism. So, **because both `Car` and `Vehicle` had a function of the same name: `drive()`**, to distinguish a call to one or the other, we must make an absolute (not relative) reference. We explicitly specify the `Vehicle` object by name, and call the `drive()` function on it.

But if we said `Vehicle.drive()`, the `this` binding for that function call would be the `Vehicle` object instead of the `Car` object (see Chapter 2), which is not what we want. So, instead we use `.call( this )` (Chapter 2) to ensure that `drive()` is executed in the context of the `Car` object.

Note: If the function name identifier for `Car.drive()` hadn't overlapped with (aka, "shadowed"; see Chapter 5) `Vehicle.drive()`, we wouldn't have been exercising "method polymorphism". So, a reference to `Vehicle.drive()` would have been copied over by the `mixin(..)` call, and we could have accessed directly with `this.drive()`. The chosen identifier overlap **shadowing** is *why* we have to use the more complex *explicit pseudo-polymorphism* approach.

In class-oriented languages, which have relative polymorphism, the linkage between `Car` and `Vehicle` is established once, at the top of the class definition, which makes for only one place to maintain such relationships.

But because of JavaScript's peculiarities, explicit pseudo-polymorphism (because of shadowing!) creates brittle manual/explicit linkage **in every single function where you need such a (pseudo-)polymorphic reference**. This can significantly increase the maintenance cost. Moreover, while explicit pseudo-polymorphism can emulate the behavior of "multiple inheritance", it only increases the complexity and brittleness.

The result of such approaches is usually more complex, harder-to-read, *and* harder-to-maintain code. **Explicit pseudo-polymorphism should be avoided wherever possible**, because the cost outweighs the benefit in most respects.

Mixing Copies

Recall the `mixin(..)` utility from above:

```js
// vastly simplified `mixin()` example:
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // only copy if not already present
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}
```

Now, let's examine how `mixin(..)` works. It iterates over the properties of `sourceObj` (`Vehicle` in our example) and if there's no matching property of that name in `targetObj` (`Car` in our example), it makes a copy. Since we're making the copy after the initial object exists, we are careful to not copy over a target property.

If we made the copies first, before specifying the `Car` specific contents, we could omit this check against `targetObj` , but that's a little more clunky and less efficient, so it's generally less preferred:

```js
// alternate mixin, less "safe" to overwrites
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        targetObj[key] = sourceObj[key];
    }

    return targetObj;
}

var Vehicle = {
    // ...
};

// first, create an empty object with
// Vehicle's stuff copied in
var Car = mixin( Vehicle, { } );

// now copy the intended contents into Car
mixin( {
    wheels: 4,

    drive: function() {
        // ...
    }
}, Car );
```

Either approach, we have explicitly copied the non-overlapping contents of `Vehicle` into `Car`. The name "mixin" comes from an alternate way of explaining the task: `Car` has `Vehicle`s contents **mixed-in**, just like you mix in chocolate chips into your favorite cookie dough.

As a result of the copy operation, `Car` will operate somewhat separately from `Vehicle`. If you add a property onto `Car`, it will not affect `Vehicle`, and vice versa.

Note: A few minor details have been skimmed over here. There are still some subtle ways the two objects can "affect" each other even after copying, such as if they both share a reference to a common object (such as an array).

Since the two objects also share references to their common functions, that means that **even manual copying of functions (aka, mixins) from one object to another doesn't *actually emulate* the real duplication from class to instance that occurs in class-oriented languages**.

JavaScript functions can't really be duplicated (in a standard, reliable way), so what you end up with instead is a **duplicated reference** to the same shared function object (functions are objects; see Chapter 3). If you modified one of the shared **function objects** (like `ignition()`) by adding properties on top of it, for instance, both `Vehicle` and `Car` would be "affected" via the shared reference.

Explicit mixins are a fine mechanism in JavaScript. But they appear more powerful than they really are. Not much benefit is *actually* derived from copying a property from one object to another, **as opposed to just defining the properties twice**, once on each object. And that's especially true given the function-object reference nuance we just mentioned.

If you explicitly mix-in two or more objects into your target object, you can **partially emulate** the behavior of "multiple inheritance", but there's no direct way to handle collisions if the same method or property is being copied from more than one source. Some developers/libraries have come up with "late binding" techniques and other exotic work-arounds, but fundamentally these "tricks" are *usually* more effort (and lesser performance!) than the pay-off.

Take care only to use explicit mixins where it actually helps make more readable code, and avoid the pattern if you find it making code that's harder to trace, or if you find it creates unnecessary or unwieldy dependencies between objects.

If it starts to get *harder* to properly use mixins than before you used them, you should probably stop using mixins. In fact, if you have to use a complex library/utility to work out all these details, it might be a sign that you're going about it the harder way, perhaps unnecessarily. In Chapter 6, we'll try to distill a simpler way that accomplishes the desired outcomes without all the fuss.

Parasitic Inheritance

A variation on this explicit mixin pattern, which is both in some ways explicit and in other ways implicit, is called "parasitic inheritance", popularized mainly by Douglas Crockford.

Here's how it can work:

```
// "Traditional JS Class" `Vehicle`
function Vehicle() {
    this.engines = 1;
}
Vehicle.prototype.ignition = function() {
    console.log( "Turning on my engine." );
};
Vehicle.prototype.drive = function() {
    this.ignition();
    console.log( "Steering and moving forward!" );
};

// "Parasitic Class" `Car`
function Car() {
    // first, `car` is a `Vehicle`
    var car = new Vehicle();

    // now, let's modify our `car` to specialize it
    car.wheels = 4;

    // save a privileged reference to `Vehicle::drive()`
    var vehDrive = car.drive;

    // override `Vehicle::drive()`
    car.drive = function() {
        vehDrive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    };

    return car;
}

var myCar = new Car();

myCar.drive();
// Turning on my engine.
// Steering and moving forward!
// Rolling on all 4 wheels!
```

As you can see, we initially make a copy of the definition from the `Vehicle` "parent class" (object), then mixin our "child class" (object) definition (preserving privileged parent-class references as needed), and pass off this composed object `car` as our child instance.

Note: when we call `new Car()`, a new object is created and referenced by `Car`s `this` reference (see Chapter 2). But since we don't use that object, and instead return our own `car` object, the initially created object is just discarded. So, `Car()` could be called without the `new` keyword, and the functionality above would be identical, but without the wasted object creation/garbage-collection.

Implicit Mixins

Implicit mixins are closely related to *explicit pseudo-polymorphism* as explained previously. As such, they come with the same caveats and warnings.

Consider this code:

```
var Something = {
    cool: function() {
        this.greeting = "Hello World";
        this.count = this.count ? this.count + 1 : 1;
    }
};

Something.cool();
Something.greeting; // "Hello World"
Something.count; // 1

var Another = {
    cool: function() {
        // implicit mixin of `Something` to `Another`
        Something.cool.call( this );
    }
};

Another.cool();
Another.greeting; // "Hello World"
Another.count; // 1 (not shared state with `Something`)
```

With `Something.cool.call( this )`, which can happen either in a "constructor" call (most common) or in a method call (shown here), we essentially "borrow" the function `Something.cool()` and call it in the context of `Another` (via its `this` binding; see Chapter 2) instead of `Something`. The end result is that the assignments that `Something.cool()` makes are applied against the `Another` object rather than the `Something` object.

So, it is said that we "mixed in" `Something`s behavior with (or into) `Another`.

While this sort of technique seems to take useful advantage of `this` rebinding functionality, it is the brittle `Something.cool.call( this )` call, which cannot be made into a relative (and thus more flexible) reference, that you should **heed with caution**. Generally, **avoid such**

constructs where possible to keep cleaner and more maintainable code.

Review (TL;DR)

Classes are a design pattern. Many languages provide syntax which enables natural class-oriented software design. JS also has a similar syntax, but it behaves **very differently** from what you're used to with classes in those other languages.

Classes mean copies.

When traditional classes are instantiated, a copy of behavior from class to instance occurs. When classes are inherited, a copy of behavior from parent to child also occurs.

Polymorphism (having different functions at multiple levels of an inheritance chain with the same name) may seem like it implies a referential relative link from child back to parent, but it's still just a result of copy behavior.

JavaScript **does not automatically** create copies (as classes imply) between objects.

The mixin pattern (both explicit and implicit) is often used to *sort of* emulate class copy behavior, but this usually leads to ugly and brittle syntax like explicit pseudo-polymorphism (`OtherObj.methodName.call(this, ...)`), which often results in harder to understand and maintain code.

Explicit mixins are also not exactly the same as class *copy*, since objects (and functions!) only have shared references duplicated, not the objects/functions duplicated themselves. Not paying attention to such nuance is the source of a variety of gotchas.

In general, faking classes in JS often sets more landmines for future coding than solving present *real* problems.

You Don't Know JS: *this* & Object Prototypes

Chapter 5: Prototypes

In Chapters 3 and 4, we mentioned the `[[Prototype]]` chain several times, but haven't said what exactly it is. We will now examine prototypes in detail.

Note: All of the attempts to emulate class-copy behavior, as described previously in Chapter 4, labeled as variations of "mixins", completely circumvent the `[[Prototype]]` chain mechanism we examine here in this chapter.

`[[Prototype]]`

Objects in JavaScript have an internal property, denoted in the specification as `[[Prototype]]`, which is simply a reference to another object. Almost all objects are given a non-`null` value for this property, at the time of their creation.

Note: We will see shortly that it *is* possible for an object to have an empty `[[Prototype]]` linkage, though this is somewhat less common.

Consider:

```
var myObject = {
    a: 2
};

myObject.a; // 2
```

What is the `[[Prototype]]` reference used for? In Chapter 3, we examined the `[[Get]]` operation that is invoked when you reference a property on an object, such as `myObject.a`. For that default `[[Get]]` operation, the first step is to check if the object itself has a property `a` on it, and if so, it's used.

Note: ES6 Proxies are outside of our discussion scope in this book (will be covered in a later book in the series!), but everything we discuss here about normal `[[Get]]` and `[[Put]]` behavior does not apply if a `Proxy` is involved.

But it's what happens if `a` **isn't** present on `myObject` that brings our attention now to the `[[Prototype]]` link of the object.

The default `[[Get]]` operation proceeds to follow the `[[Prototype]]` **link** of the object if it cannot find the requested property on the object directly.

```
var anotherObject = {
    a: 2
};

// create an object linked to `anotherObject`
var myObject = Object.create( anotherObject );

myObject.a; // 2
```

Note: We will explain what `Object.create(..)` does, and how it operates, shortly. For now, just assume it creates an object with the `[[Prototype]]` linkage we're examining to the object specified.

So, we have `myObject` that is now `[[Prototype]]` linked to `anotherObject`. Clearly `myObject.a` doesn't actually exist, but nevertheless, the property access succeeds (being found on `anotherObject` instead) and indeed finds the value `2`.

But, if `a` weren't found on `anotherObject` either, its `[[Prototype]]` chain, if non-empty, is again consulted and followed.

This process continues until either a matching property name is found, or the `[[Prototype]]` chain ends. If no matching property is *ever* found by the end of the chain, the return result from the `[[Get]]` operation is `undefined`.

Similar to this `[[Prototype]]` chain look-up process, if you use a `for..in` loop to iterate over an object, any property that can be reached via its chain (and is also `enumerable` -- see Chapter 3) will be enumerated. If you use the `in` operator to test for the existence of a property on an object, `in` will check the entire chain of the object (regardless of *enumerability*).

```
var anotherObject = {
    a: 2
};

// create an object linked to `anotherObject`
var myObject = Object.create( anotherObject );

for (var k in myObject) {
    console.log("found: " + k);
}
// found: a

("a" in myObject); // true
```

So, the `[[Prototype]]` chain is consulted, one link at a time, when you perform property look-ups in various fashions. The look-up stops once the property is found or the chain ends.

`Object.prototype`

But *where* exactly does the `[[Prototype]]` chain "end"?

The top-end of every *normal* `[[Prototype]]` chain is the built-in `Object.prototype`. This object includes a variety of common utilities used all over JS, because all normal (built-in, not host-specific extension) objects in JavaScript "descend from" (aka, have at the top of their `[[Prototype]]` chain) the `Object.prototype` object.

Some utilities found here you may be familiar with include `.toString()` and `.valueOf()`. In Chapter 3, we introduced another: `.hasOwnProperty(..)`. And yet another function on `Object.prototype` you may not be familiar with, but which we'll address later in this chapter, is `.isPrototypeOf(..)`.

Setting & Shadowing Properties

Back in Chapter 3, we mentioned that setting properties on an object was more nuanced than just adding a new property to the object or changing an existing property's value. We will now revisit this situation more completely.

```
myObject.foo = "bar";
```

If the `myObject` object already has a normal data accessor property called `foo` directly present on it, the assignment is as simple as changing the value of the existing property.

If `foo` is not already present directly on `myObject`, the `[[Prototype]]` chain is traversed, just like for the `[[Get]]` operation. If `foo` is not found anywhere in the chain, the property `foo` is added directly to `myObject` with the specified value, as expected.

However, if `foo` is already present somewhere higher in the chain, nuanced (and perhaps surprising) behavior can occur with the `myObject.foo = "bar"` assignment. We'll examine that more in just a moment.

If the property name `foo` ends up both on `myObject` itself and at a higher level of the `[[Prototype]]` chain that starts at `myObject`, this is called *shadowing*. The `foo` property directly on `myObject` *shadows* any `foo` property which appears higher in the chain, because the `myObject.foo` look-up would always find the `foo` property that's lowest in the chain.

As we just hinted, shadowing `foo` on `myObject` is not as simple as it may seem. We will now examine three scenarios for the `myObject.foo = "bar"` assignment when `foo` is **not** already on `myObject` directly, but **is** at a higher level of `myObject`'s `[[Prototype]]` chain:

1. If a normal data accessor (see Chapter 3) property named `foo` is found anywhere higher on the `[[Prototype]]` chain, **and it's not marked as read-only (`writable:false`)** then a new property called `foo` is added directly to `myObject`, resulting in a **shadowed property**.
2. If a `foo` is found higher on the `[[Prototype]]` chain, but it's marked as **read-only (`writable:false`)**, then both the setting of that existing property as well as the creation of the shadowed property on `myObject` **are disallowed**. If the code is running in `strict mode`, an error will be thrown. Otherwise, the setting of the property value will silently be ignored. Either way, **no shadowing occurs**.
3. If a `foo` is found higher on the `[[Prototype]]` chain and it's a setter (see Chapter 3), then the setter will always be called. No `foo` will be added to (aka, shadowed on) `myObject`, nor will the `foo` setter be redefined.

Most developers assume that assignment of a property (`[[Put]]`) will always result in shadowing if the property already exists higher on the `[[Prototype]]` chain, but as you can see, that's only true in one (#1) of the three situations just described.

If you want to shadow `foo` in cases #2 and #3, you cannot use `=` assignment, but must instead use `Object.defineProperty(..)` (see Chapter 3) to add `foo` to `myObject`.

Note: Case #2 may be the most surprising of the three. The presence of a *read-only* property prevents a property of the same name being implicitly created (shadowed) at a lower level of a `[[Prototype]]` chain. The reason for this restriction is primarily to reinforce the illusion of class-inherited properties. If you think of the `foo` at a higher level of the chain as having been inherited (copied down) to `myObject`, then it makes sense to enforce the non-writable nature of that `foo` property on `myObject`. If you however separate the illusion

from the fact, and recognize that no such inheritance copying *actually* occured (see Chapters 4 and 5), it's a little unnatural that `myObject` would be prevented from having a `foo` property just because some other object had a non-writable `foo` on it. It's even stranger that this restriction only applies to `=` assignment, but is not enforced when using `Object.defineProperty(..)`.

Shadowing with **methods** leads to ugly *explicit pseudo-polymorphism* (see Chapter 4) if you need to delegate between them. Usually, shadowing is more complicated and nuanced than it's worth, **so you should try to avoid it if possible**. See Chapter 6 for an alternative design pattern, which among other things discourages shadowing in favor of cleaner alternatives.

Shadowing can even occur implicitly in subtle ways, so care must be taken if trying to avoid it. Consider:

```
var anotherObject = {
    a: 2
};

var myObject = Object.create( anotherObject );

anotherObject.a; // 2
myObject.a; // 2

anotherObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "a" ); // false

myObject.a++; // oops, implicit shadowing!

anotherObject.a; // 2
myObject.a; // 3

myObject.hasOwnProperty( "a" ); // true
```

Though it may appear that `myObject.a++` should (via delegation) look-up and just increment the `anotherObject.a` property itself *in place*, instead the `++` operation corresponds to `myObject.a = myObject.a + 1`. The result is `[[Get]]` looking up `a` property via `[[Prototype]]` to get the current value `2` from `anotherObject.a`, incrementing the value by one, then `[[Put]]` assigning the `3` value to a new shadowed property `a` on `myObject`. Oops!

Be very careful when dealing with delegated properties that you modify. If you wanted to increment `anotherObject.a`, the only proper way is `anotherObject.a++`.

"Class"

At this point, you might be wondering: "*Why* does one object need to link to another object?" What's the real benefit? That is a very appropriate question to ask, but we must first understand what `[[Prototype]]` is **not** before we can fully understand and appreciate what it *is* and how it's useful.

As we explained in Chapter 4, in JavaScript, there are no abstract patterns/blueprints for objects called "classes" as there are in class-oriented languages. JavaScript **just** has objects.

In fact, JavaScript is **almost unique** among languages as perhaps the only language with the right to use the label "object oriented", because it's one of a very short list of languages where an object can be created directly, without a class at all.

In JavaScript, classes can't (being that they don't exist!) describe what an object can do. The object defines its own behavior directly. **There's *just* the object.**

"Class" Functions

There's a peculiar kind of behavior in JavaScript that has been shamelessly abused for years to *hack* something that *looks* like "classes". We'll examine this approach in detail.

The peculiar "sort-of class" behavior hinges on a strange characteristic of functions: all functions by default get a public, non-enumerable (see Chapter 3) property on them called `prototype`, which points at an otherwise arbitrary object.

```
function Foo() {
    // ...
}

Foo.prototype; // { }
```

This object is often called "Foo's prototype", because we access it via an unfortunately-named `Foo.prototype` property reference. However, that terminology is hopelessly destined to lead us into confusion, as we'll see shortly. Instead, I will call it "the object formerly known as Foo's prototype". Just kidding. How about: "object arbitrarily labeled 'Foo dot prototype'"?

Whatever we call it, what exactly is this object?

The most direct way to explain it is that each object created from calling `new Foo()` (see Chapter 2) will end up (somewhat arbitrarily) `[[Prototype]]`-linked to this "Foo dot prototype" object.

Let's illustrate:

```
function Foo() {
    // ...
}

var a = new Foo();

Object.getPrototypeOf( a ) === Foo.prototype; // true
```

When `a` is created by calling `new Foo()`, one of the things (see Chapter 2 for all *four* steps) that happens is that `a` gets an internal `[[Prototype]]` link to the object that `Foo.prototype` is pointing at.

Stop for a moment and ponder the implications of that statement.

In class-oriented languages, multiple **copies** (aka, "instances") of a class can be made, like stamping something out from a mold. As we saw in Chapter 4, this happens because the process of instantiating (or inheriting from) a class means, "copy the behavior plan from that class into a physical object", and this is done again for each new instance.

But in JavaScript, there are no such copy-actions performed. You don't create multiple instances of a class. You can create multiple objects that `[[Prototype]]` *link* to a common object. But by default, no copying occurs, and thus these objects don't end up totally separate and disconnected from each other, but rather, quite ***linked***.

`new Foo()` results in a new object (we called it `a`), and **that** new object `a` is internally `[[Prototype]]` linked to the `Foo.prototype` object.

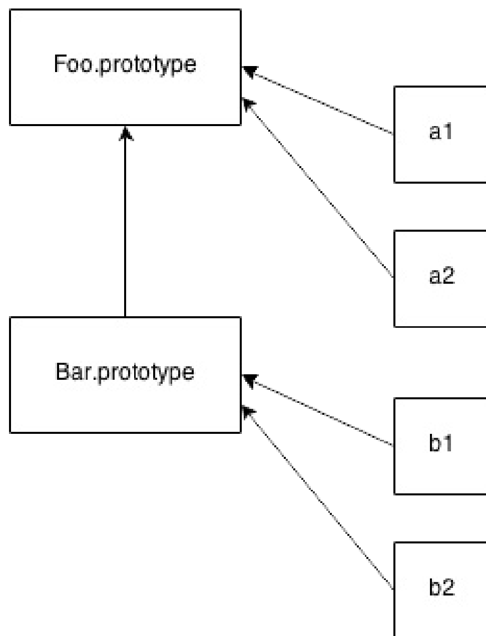
We end up with two objects, linked to each other. That's *it*. We didn't instantiate a class. We certainly didn't do any copying of behavior from a "class" into a concrete object. We just caused two objects to be linked to each other.

In fact, the secret, which eludes most JS developers, is that the `new Foo()` function calling had really almost nothing *direct* to do with the process of creating the link. **It was sort of an accidental side-effect.** `new Foo()` is an indirect, round-about way to end up with what we want: **a new object linked to another object**.

Can we get what we want in a more *direct* way? **Yes!** The hero is `Object.create(..)`. But we'll get to that in a little bit.

What's in a name?

In JavaScript, we don't make *copies* from one object ("class") to another ("instance"). We make *links* between objects. For the `[[Prototype]]` mechanism, visually, the arrows move from right to left, and from bottom to top.



This mechanism is often called "prototypal inheritance" (we'll explore the code in detail shortly), which is commonly said to be the dynamic-language version of "classical inheritance". It's an attempt to piggy-back on the common understanding of what "inheritance" means in the class-oriented world, but *tweak* (**read: pave over**) the understood semantics, to fit dynamic scripting.

The word "inheritance" has a very strong meaning (see Chapter 4), with plenty of mental precedent. Merely adding "prototypal" in front to distinguish the *actually nearly opposite* behavior in JavaScript has left in its wake nearly two decades of miry confusion.

I like to say that sticking "prototypal" in front "inheritance" to drastically reverse its actual meaning is like holding an orange in one hand, an apple in the other, and insisting on calling the apple a "red orange". No matter what confusing label I put in front of it, that doesn't change the *fact* that one fruit is an apple and the other is an orange.

The better approach is to plainly call an apple an apple -- to use the most accurate and direct terminology. That makes it easier to understand both their similarities and their **many differences**, because we all have a simple, shared understanding of what "apple" means.

Because of the confusion and conflation of terms, I believe the label "prototypal inheritance" itself (and trying to mis-apply all its associated class-orientation terminology, like "class", "constructor", "instance", "polymorphism", etc) has done **more harm than good** in explaining how JavaScript's mechanism *really* works.

"Inheritance" implies a *copy* operation, and JavaScript doesn't copy object properties (natively, by default). Instead, JS creates a link between two objects, where one object can essentially *delegate* property/function access to another object. "Delegation" (see Chapter 6) is a much more accurate term for JavaScript's object-linking mechanism.

Another term which is sometimes thrown around in JavaScript is "differential inheritance". The idea here is that we describe an object's behavior in terms of what is *different* from a more general descriptor. For example, you explain that a car is a kind of vehicle, but one that has exactly 4 wheels, rather than re-describing all the specifics of what makes up a general vehicle (engine, etc).

If you try to think of any given object in JS as the sum total of all behavior that is *available* via delegation, and **in your mind you flatten** all that behavior into one tangible *thing*, then you can (sorta) see how "differential inheritance" might fit.

But just like with "prototypal inheritance", "differential inheritance" pretends that your mental model is more important than what is physically happening in the language. It overlooks the fact that object `B` is not actually differentially constructed, but is instead built with specific characteristics defined, alongside "holes" where nothing is defined. It is in these "holes" (gaps in, or lack of, definition) that delegation *can* take over and, on the fly, "fill them in" with delegated behavior.

The object is not, by native default, flattened into the single differential object, **through copying**, that the mental model of "differential inheritance" implies. As such, "differential inheritance" is just not as natural a fit for describing how JavaScript's `[[Prototype]]` mechanism actually works.

You *can choose* to prefer the "differential inheritance" terminology and mental model, as a matter of taste, but there's no denying the fact that it *only* fits the mental acrobatics in your mind, not the physical behavior in the engine.

"Constructors"

Let's go back to some earlier code:

```
function Foo() {
    // ...
}

var a = new Foo();
```

What exactly leads us to think `Foo` is a "class"?

For one, we see the use of the `new` keyword, just like class-oriented languages do when they construct class instances. For another, it appears that we are in fact executing a *constructor* method of a class, because `Foo()` is actually a method that gets called, just like how a real class's constructor gets called when you instantiate that class.

To further the confusion of "constructor" semantics, the arbitrarily labeled `Foo.prototype` object has another trick up its sleeve. Consider this code:

```
function Foo() {
    // ...
}

Foo.prototype.constructor === Foo; // true

var a = new Foo();
a.constructor === Foo; // true
```

The `Foo.prototype` object by default (at declaration time on line 1 of the snippet!) gets a public, non-enumerable (see Chapter 3) property called `.constructor`, and this property is a reference back to the function (`Foo` in this case) that the object is associated with. Moreover, we see that object `a` created by the "constructor" call `new Foo()` *seems* to also have a property on it called `.constructor` which similarly points to "the function which created it".

Note: This is not actually true. `a` has no `.constructor` property on it, and though `a.constructor` does in fact resolve to the `Foo` function, "constructor" **does not actually mean** "was constructed by", as it appears. We'll explain this strangeness shortly.

Oh, yeah, also... by convention in the JavaScript world, "class"es are named with a capital letter, so the fact that it's `Foo` instead of `foo` is a strong clue that we intend it to be a "class". That's totally obvious to you, right!?

Note: This convention is so strong that many JS linters actually *complain* if you call `new` on a method with a lowercase name, or if we don't call `new` on a function that happens to start with a capital letter. That sort of boggles the mind that we struggle so much to get (fake) "class-orientation" *right* in JavaScript that we create linter rules to ensure we use capital letters, even though the capital letter doesn't mean ***anything* at all** to the JS engine.

Constructor Or Call?

In the above snippet, it's tempting to think that `Foo` is a "constructor", because we call it with `new` and we observe that it "constructs" an object.

In reality, `Foo` is no more a "constructor" than any other function in your program. Functions themselves are **not** constructors. However, when you put the `new` keyword in front of a normal function call, that makes that function call a "constructor call". In fact, `new` sort of hijacks any normal function and calls it in a fashion that constructs an object, **in addition to whatever else it was going to do**.

For example:

```
function NothingSpecial() {
    console.log( "Don't mind me!" );
}

var a = new NothingSpecial();
// "Don't mind me!"

a; // {}
```

`NothingSpecial` is just a plain old normal function, but when called with `new`, it *constructs* an object, almost as a side-effect, which we happen to assign to `a`. The **call** was a *constructor call*, but `NothingSpecial` is not, in and of itself, a *constructor*.

In other words, in JavaScript, it's most appropriate to say that a "constructor" is **any function called with the `new` keyword** in front of it.

Functions aren't constructors, but function calls are "constructor calls" if and only if `new` is used.

Mechanics

Are *those* the only common triggers for ill-fated "class" discussions in JavaScript?

Not quite. JS developers have strived to simulate as much as they can of class-orientation:

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

var a = new Foo( "a" );
var b = new Foo( "b" );

a.myName(); // "a"
b.myName(); // "b"
```

This snippet shows two additional "class-orientation" tricks in play:

1. `this.name = name`: adds the `.name` property onto each object (`a` and `b`, respectively; see Chapter 2 about `this` binding), similar to how class instances encapsulate data values.

2. `Foo.prototype.myName = ...` : perhaps the more interesting technique, this adds a property (function) to the `Foo.prototype` object. Now, `a.myName()` works, but perhaps surprisingly. How?

In the above snippet, it's strongly tempting to think that when `a` and `b` are created, the properties/functions on the `Foo.prototype` object are *copied* over to each of `a` and `b` objects. **However, that's not what happens.**

At the beginning of this chapter, we explained the `[[Prototype]]` link, and how it provides the fall-back look-up steps if a property reference isn't found directly on an object, as part of the default `[[Get]]` algorithm.

So, by virtue of how they are created, `a` and `b` each end up with an internal `[[Prototype]]` linkage to `Foo.prototype`. When `myName` is not found on `a` or `b`, respectively, it's instead found (through delegation, see Chapter 6) on `Foo.prototype`.

"Constructor" Redux

Recall the discussion from earlier about the `.constructor` property, and how it *seems* like `a.constructor === Foo` being true means that `a` has an actual `.constructor` property on it, pointing at `Foo`? **Not correct.**

This is just unfortunate confusion. In actuality, the `.constructor` reference is also *delegated* up to `Foo.prototype`, which **happens to**, by default, have a `.constructor` that points at `Foo`.

It *seems* awfully convenient that an object `a` "constructed by" `Foo` would have access to a `.constructor` property that points to `Foo`. But that's nothing more than a false sense of security. It's a happy accident, almost tangentially, that `a.constructor` *happens* to point at `Foo` via this default `[[Prototype]]` delegation. There's actually several ways that the ill-fated assumption of `.constructor` meaning "was constructed by" can come back to bite you.

For one, the `.constructor` property on `Foo.prototype` is only there by default on the object created when `Foo` the function is declared. If you create a new object, and replace a function's default `.prototype` object reference, the new object will not by default magically get a `.constructor` on it.

Consider:

```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // create a new prototype object

var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!
```

`Object(..)` didn't "construct" `a1` did it? It sure seems like `Foo()` "constructed" it. Many developers think of `Foo()` as doing the construction, but where everything falls apart is when you think "constructor" means "was constructed by", because by that reasoning, `a1.constructor` should be `Foo`, but it isn't!

What's happening? `a1` has no `.constructor` property, so it delegates up the `[[Prototype]]` chain to `Foo.prototype`. But that object doesn't have a `.constructor` either (like the default `Foo.prototype` object would have had!), so it keeps delegating, this time up to `Object.prototype`, the top of the delegation chain. *That* object indeed has a `.constructor` on it, which points to the built-in `Object(..)` function.

Misconception, busted.

Of course, you can add `.constructor` back to the `Foo.prototype` object, but this takes manual work, especially if you want to match native behavior and have it be non-enumerable (see Chapter 3).

For example:

```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // create a new prototype object

// Need to properly "fix" the missing `.constructor`
// property on the new object serving as `Foo.prototype`.
// See Chapter 3 for `defineProperty(..)`.
Object.defineProperty( Foo.prototype, "constructor" , {
    enumerable: false,
    writable: true,
    configurable: true,
    value: Foo    // point `.constructor` at `Foo`
} );
```

That's a lot of manual work to fix `.constructor`. Moreover, all we're really doing is perpetuating the misconception that "constructor" means "was constructed by". That's an *expensive* illusion.

The fact is, `.constructor` on an object arbitrarily points, by default, at a function who, reciprocally, has a reference back to the object -- a reference which it calls `.prototype`. The words "constructor" and "prototype" only have a loose default meaning that might or might not hold true later. The best thing to do is remind yourself, "constructor does not mean constructed by".

`.constructor` is not a magic immutable property. It *is* non-enumerable (see snippet above), but its value is writable (can be changed), and moreover, you can add or overwrite (intentionally or accidentally) a property of the name `constructor` on any object in any `[[Prototype]]` chain, with any value you see fit.

By virtue of how the `[[Get]]` algorithm traverses the `[[Prototype]]` chain, a `.constructor` property reference found anywhere may resolve quite differently than you'd expect.

See how arbitrary its meaning actually is?

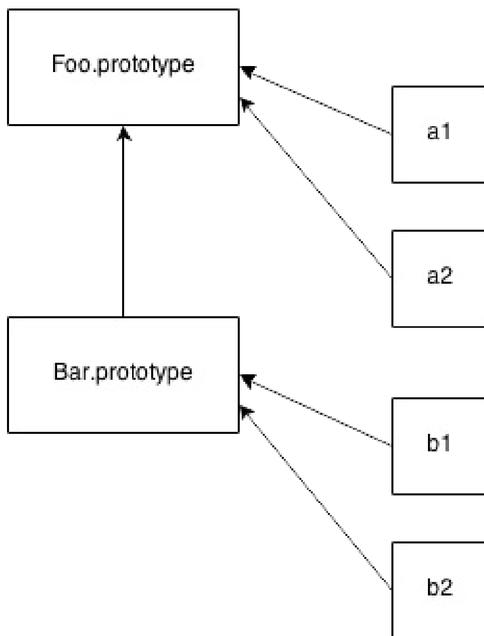
The result? Some arbitrary object-property reference like `a1.constructor` cannot actually be *trusted* to be the assumed default function reference. Moreover, as we'll see shortly, just by simple omission, `a1.constructor` can even end up pointing somewhere quite surprising and insensible.

`a1.constructor` is extremely unreliable, and an unsafe reference to rely upon in your code. **Generally, such references should be avoided where possible.**

"(Prototypal) Inheritance"

We've seen some approximations of "class" mechanics as typically hacked into JavaScript programs. But JavaScript "class"es would be rather hollow if we didn't have an approximation of "inheritance".

Actually, we've already seen the mechanism which is commonly called "prototypal inheritance" at work when `a` was able to "inherit from" `Foo.prototype`, and thus get access to the `myName()` function. But we traditionally think of "inheritance" as being a relationship between two "classes", rather than between "class" and "instance".



Recall this figure from earlier, which shows not only delegation from an object (aka, "instance") `a1` to object `Foo.prototype`, but from `Bar.prototype` to `Foo.prototype`, which somewhat resembles the concept of Parent-Child class inheritance. *Resembles*, except of course for the direction of the arrows, which show these are delegation links rather than copy operations.

And, here's the typical "prototype style" code that creates such links:

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

function Bar(name,label) {
    Foo.call( this, name );
    this.label = label;
}

// here, we make a new `Bar.prototype`
// linked to `Foo.prototype`
Bar.prototype = Object.create( Foo.prototype );

// Beware! Now `Bar.prototype.constructor` is gone,
// and might need to be manually "fixed" if you're
// in the habit of relying on such properties!

Bar.prototype.myLabel = function() {
    return this.label;
};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"
```

Note: To understand why `this` points to `a` in the above code snippet, see Chapter 2.

The important part is `Bar.prototype = Object.create( Foo.prototype )`. `Object.create(..)` *creates* a "new" object out of thin air, and links that new object's internal `[[Prototype]]` to the object you specify (`Foo.prototype` in this case).

In other words, that line says: "make a *new* 'Bar dot prototype' object that's linked to 'Foo dot prototype'."

When `function Bar() { .. }` is declared, `Bar`, like any other function, has a `.prototype` link to its default object. But *that* object is not linked to `Foo.prototype` like we want. So, we create a *new* object that *is* linked as we want, effectively throwing away the original incorrectly-linked object.

Note: A common mis-conception/confusion here is that either of the following approaches would *also* work, but they do not work as you'd expect:

```
// doesn't work like you want!
Bar.prototype = Foo.prototype;

// works kinda like you want, but with
// side-effects you probably don't want :(
Bar.prototype = new Foo();
```

`Bar.prototype = Foo.prototype` doesn't create a new object for `Bar.prototype` to be linked to. It just makes `Bar.prototype` be another reference to `Foo.prototype`, which effectively links `Bar` directly to **the same object as** `Foo` links to: `Foo.prototype`. This means when you start assigning, like `Bar.prototype.myLabel = ...`, you're modifying **not a separate object** but *the* shared `Foo.prototype` object itself, which would affect any objects linked to `Foo.prototype`. This is almost certainly not what you want. If it *is* what you want, then you likely don't need `Bar` at all, and should just use only `Foo` and make your code simpler.

`Bar.prototype = new Foo()` **does in fact** create a new object which is duly linked to `Foo.prototype` as we'd want. But, it uses the `Foo(..)` "constructor call" to do it. If that function has any side-effects (such as logging, changing state, registering against other objects, **adding data properties to `this`**, etc.), those side-effects happen at the time of this linking (and likely against the wrong object!), rather than only when the eventual `Bar()` "descendents" are created, as would likely be expected.

So, we're left with using `Object.create(..)` to make a new object that's properly linked, but without having the side-effects of calling `Foo(..)`. The slight downside is that we have to create a new object, throwing the old one away, instead of modifying the existing default object we're provided.

It would be *nice* if there was a standard and reliable way to modify the linkage of an existing object. Prior to ES6, there's a non-standard and not fully-cross-browser way, via the `.__proto__` property, which is settable. ES6 adds a `Object.setPrototypeOf(..)` helper utility, which does the trick in a standard and predictable way.

Compare the pre-ES6 and ES6-standardized techniques for linking `Bar.prototype` to `Foo.prototype`, side-by-side:

```
// pre-ES6
// throws away default existing `Bar.prototype`
Bar.prototype = Object.create( Foo.prototype );

// ES6+
// modifies existing `Bar.prototype`
Object.setPrototypeOf( Bar.prototype, Foo.prototype );
```

Ignoring the slight performance disadvantage (throwing away an object that's later garbage collected) of the `Object.create(..)` approach, it's a little bit shorter and may be perhaps a little easier to read than the ES6+ approach. But it's probably a syntactic wash either way.

Inspecting "Class" Relationships

What if you have an object like `a` and want to find out what object (if any) it delegates to? Inspecting an instance (just an object in JS) for its inheritance ancestry (delegation linkage in JS) is often called *introspection* (or *reflection*) in traditional class-oriented environments.

Consider:

```
function Foo() {
    // ...
}

Foo.prototype.blah = ...;

var a = new Foo();
```

How do we then introspect `a` to find out its "ancestry" (delegation linkage)? The first approach embraces the "class" confusion:

```
a instanceof Foo; // true
```

The `instanceof` operator takes a plain object as its left-hand operand and a **function** as its right-hand operand. The question `instanceof` answers is: **in the entire `[[Prototype]]` chain of `a`, does the object arbitrarily pointed to by `Foo.prototype` ever appear?**

Unfortunately, this means that you can only inquire about the "ancestry" of some object (`a`) if you have some **function** (`Foo` , with its attached `.prototype` reference) to test with. If you have two arbitrary objects, say `a` and `b` , and want to find out if *the objects* are related to each other through a `[[Prototype]]` chain, `instanceof` alone can't help.

Note: If you use the built-in `.bind(..)` utility to make a hard-bound function (see Chapter 2), the function created will not have a `.prototype` property. Using `instanceof` with such a function transparently substitutes the `.prototype` of the *target function* that the hard-bound function was created from.

It's fairly uncommon to use hard-bound functions as "constructor calls", but if you do, it will behave as if the original *target function* was invoked instead, which means that using `instanceof` with a hard-bound function also behaves according to the original function.

This snippet illustrates the ridiculousness of trying to reason about relationships between **two objects** using "class" semantics and `instanceof` :

```
// helper utility to see if `o1` is
// related to (delegates to) `o2`
function isRelatedTo(o1, o2) {
    function F(){}
    F.prototype = o2;
    return o1 instanceof F;
}

var a = {};
var b = Object.create( a );

isRelatedTo( b, a ); // true
```

Inside `isRelatedTo(..)` , we borrow a throw-away function `F` , reassign its `.prototype` to arbitrarily point to some object `o2` , then ask if `o1` is an "instance of" `F` . Obviously `o1` isn't *actually* inherited or descended or even constructed from `F` , so it should be clear why this kind of exercise is silly and confusing. **The problem comes down to the awkwardness of class semantics forced upon JavaScript**, in this case as revealed by the indirect semantics of `instanceof` .

The second, and much cleaner, approach to `[[Prototype]]` reflection is:

```
Foo.prototype.isPrototypeOf( a ); // true
```

Notice that in this case, we don't really care about (or even *need*) `Foo` , we just need an **object** (in our case, arbitrarily labeled `Foo.prototype`) to test against another **object**. The question `isPrototypeOf(..)` answers is: **in the entire `[[Prototype]]` chain of `a` , does `Foo.prototype` ever appear?**

Same question, and exact same answer. But in this second approach, we don't actually need the indirection of referencing a **function** (`Foo`) whose `.prototype` property will automatically be consulted.

We *just need* two **objects** to inspect a relationship between them. For example:

```
// Simply: does `b` appear anywhere in
// `c`s [[Prototype]] chain?
b.isPrototypeOf( c );
```

Notice, this approach doesn't require a function ("class") at all. It just uses object references directly to `b` and `c`, and inquires about their relationship. In other words, our `isRelatedTo(..)` utility above is built-in to the language, and it's called `isPrototypeOf(..)`.

We can also directly retrieve the `[[Prototype]]` of an object. As of ES5, the standard way to do this is:

```
Object.getPrototypeOf( a );
```

And you'll notice that object reference is what we'd expect:

```
Object.getPrototypeOf( a ) === Foo.prototype; // true
```

Most browsers (not all!) have also long supported a non-standard alternate way of accessing the internal `[[Prototype]]`:

```
a.__proto__ === Foo.prototype; // true
```

The strange `.__proto__` (not standardized until ES6!) property "magically" retrieves the internal `[[Prototype]]` of an object as a reference, which is quite helpful if you want to directly inspect (or even traverse: `.__proto__.__proto__...`) the chain.

Just as we saw earlier with `.constructor`, `.__proto__` doesn't actually exist on the object you're inspecting (`a` in our running example). In fact, it exists (non-enumerable; see Chapter 2) on the built-in `Object.prototype`, along with the other common utilities (`.toString()`, `.isPrototypeOf(..)`, etc).

Moreover, `.__proto__` looks like a property, but it's actually more appropriate to think of it as a getter/setter (see Chapter 3).

Roughly, we could envision `.__proto__` implemented (see Chapter 3 for object property definitions) like this:

```
Object.defineProperty( Object.prototype, "__proto__", {
    get: function() {
        return Object.getPrototypeOf( this );
    },
    set: function(o) {
        // setPrototypeOf(..) as of ES6
        Object.setPrototypeOf( this, o );
        return o;
    }
} );
```

So, when we access (retrieve the value of) `a.__proto__`, it's like calling `a.__proto__()` (calling the getter function). *That* function call has `a` as its `this` even though the getter function exists on the `Object.prototype` object (see Chapter 2 for `this` binding rules), so it's just like saying `Object.getPrototypeOf( a )`.

`.__proto__` is also a settable property, just like using ES6's `Object.setPrototypeOf(..)` shown earlier. However, generally you **should not change the `[[Prototype]]` of an existing object**.

There are some very complex, advanced techniques used deep in some frameworks that allow tricks like "subclassing" an `Array`, but this is commonly frowned on in general programming practice, as it usually leads to *much* harder to understand/maintain code.

Note: As of ES6, the `class` keyword will allow something that approximates "subclassing" of built-in's like `Array`. See Appendix A for discussion of the `class` syntax added in ES6.

The only other narrow exception (as mentioned earlier) would be setting the `[[Prototype]]` of a default function's `.prototype` object to reference some other object (besides `Object.prototype`). That would avoid replacing that default object entirely with a new linked object. Otherwise, **it's best to treat object `[[Prototype]]` linkage as a read-only characteristic** for ease of reading your code later.

Note: The JavaScript community unofficially coined a term for the double-underscore, specifically the leading one in properties like `__proto__`: "dunder". So, the "cool kids" in JavaScript would generally pronounce `__proto__` as "dunder proto".

Object Links

As we've now seen, the `[[Prototype]]` mechanism is an internal link that exists on one object which references some other object.

This linkage is (primarily) exercised when a property/method reference is made against the first object, and no such property/method exists. In that case, the `[[Prototype]]` linkage tells the engine to look for the property/method on the linked-to object. In turn, if that object cannot fulfill the look-up, its `[[Prototype]]` is followed, and so on. This series of links between objects forms what is called the "prototype chain".

`Create()`ing Links

We've thoroughly debunked why JavaScript's `[[Prototype]]` mechanism is **not** like *classes*, and we've seen how it instead creates **links** between proper objects.

What's the point of the `[[Prototype]]` mechanism? Why is it so common for JS developers to go to so much effort (emulating classes) in their code to wire up these linkages?

Remember we said much earlier in this chapter that `Object.create(..)` would be a hero? Now, we're ready to see how.

```
var foo = {
    something: function() {
        console.log( "Tell me something good..." );
    }
};

var bar = Object.create( foo );

bar.something(); // Tell me something good...
```

`Object.create(..)` creates a new object (`bar`) linked to the object we specified (`foo`), which gives us all the power (delegation) of the `[[Prototype]]` mechanism, but without any of the unnecessary complication of `new` functions acting as classes and constructor calls, confusing `.prototype` and `.constructor` references, or any of that extra stuff.

Note: `Object.create(null)` creates an object that has an empty (aka, `null`) `[[Prototype]]` linkage, and thus the object can't delegate anywhere. Since such an object has no prototype chain, the `instanceof` operator (explained earlier) has nothing to check, so it will always return `false` . These special empty- `[[Prototype]]` objects are often called "dictionaries" as they are typically used purely for storing data in properties, mostly because they have no possible surprise effects from any delegated properties/functions on the `[[Prototype]]` chain, and are thus purely flat data storage.

We don't *need* classes to create meaningful relationships between two objects. The only thing we should **really care about** is objects linked together for delegation, and `Object.create(..)` gives us that linkage without all the class cruft.

`Object.create()` Polyfilled

`Object.create(..)` was added in ES5. You may need to support pre-ES5 environments (like older IE's), so let's take a look at a simple **partial** polyfill for `Object.create(..)` that gives us the capability that we need even in those older JS environments:

```
if (!Object.create) {
    Object.create = function(o) {
        function F(){}
        F.prototype = o;
        return new F();
    };
}
```

This polyfill works by using a throw-away `F` function and overriding its `.prototype` property to point to the object we want to link to. Then we use `new F()` construction to make a new object that will be linked as we specified.

This usage of `Object.create(..)` is by far the most common usage, because it's the part that *can be* polyfilled. There's an additional set of functionality that the standard ES5 built-in `Object.create(..)` provides, which is **not polyfillable** for pre-ES5. As such, this capability is far-less commonly used. For completeness sake, let's look at that additional functionality:

```
var anotherObject = {
    a: 2
};

var myObject = Object.create( anotherObject, {
    b: {
        enumerable: false,
        writable: true,
        configurable: false,
        value: 3
    },
    c: {
        enumerable: true,
        writable: false,
        configurable: false,
        value: 4
    }
} );

myObject.hasOwnProperty( "a" ); // false
myObject.hasOwnProperty( "b" ); // true
myObject.hasOwnProperty( "c" ); // true

myObject.a; // 2
myObject.b; // 3
myObject.c; // 4
```

The second argument to `Object.create(..)` specifies property names to add to the newly created object, via declaring each new property's *property descriptor* (see Chapter 3). Because polyfilling property descriptors into pre-ES5 is not possible, this additional

functionality on `Object.create(..)` also cannot be polyfilled.

The vast majority of usage of `Object.create(..)` uses the polyfill-safe subset of functionality, so most developers are fine with using the **partial polyfill** in pre-ES5 environments.

Some developers take a much stricter view, which is that no function should be polyfilled unless it can be *fully* polyfilled. Since `Object.create(..)` is one of those partial-polyfill'able utilities, this narrower perspective says that if you need to use any of the functionality of `Object.create(..)` in a pre-ES5 environment, instead of polyfilling, you should use a custom utility, and stay away from using the name `Object.create` entirely. You could instead define your own utility, like:

```
function createAndLinkObject(o) {
    function F(){}
    F.prototype = o;
    return new F();
}

var anotherObject = {
    a: 2
};

var myObject = createAndLinkObject( anotherObject );

myObject.a; // 2
```

I do not share this strict opinion. I fully endorse the common partial-polyfill of `Object.create(..)` as shown above, and using it in your code even in pre-ES5. I'll leave it to you to make your own decision.

Links As Fallbacks?

It may be tempting to think that these links between objects *primarily* provide a sort of fallback for "missing" properties or methods. While that may be an observed outcome, I don't think it represents the right way of thinking about `[[Prototype]]`.

Consider:

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.cool(); // "cool!"
```

That code will work by virtue of `[[Prototype]]`, but if you wrote it that way so that `anotherObject` was acting as a fallback **just in case** `myObject` couldn't handle some property/method that some developer may try to call, odds are that your software is going to be a bit more "magical" and harder to understand and maintain.

That's not to say there aren't cases where fallbacks are an appropriate design pattern, but it's not very common or idiomatic in JS, so if you find yourself doing so, you might want to take a step back and reconsider if that's really appropriate and sensible design.

Note: In ES6, an advanced functionality called `Proxy` is introduced which can provide something of a "method not found" type of behavior. `Proxy` is beyond the scope of this book, but will be covered in detail in a later book in the *"You Don't Know JS"* series.

Don't miss an important but nuanced point here.

Designing software where you intend for a developer to, for instance, call `myObject.cool()` and have that work even though there is no `cool()` method on `myObject` introduces some "magic" into your API design that can be surprising for future developers who maintain your software.

You can however design your API with less "magic" to it, but still take advantage of the power of `[[Prototype]]` linkage.

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.doCool = function() {
    this.cool(); // internal delegation!
};

myObject.doCool(); // "cool!"
```

Here, we call `myObject.doCool()`, which is a method that *actually exists* on `myObject`, making our API design more explicit (less "magical"). *Internally*, our implementation follows the **delegation design pattern** (see Chapter 6), taking advantage of `[[Prototype]]` delegation to `anotherObject.cool()`.

In other words, delegation will tend to be less surprising/confusing if it's an internal implementation detail rather than plainly exposed in your API interface design. We will expound on **delegation** in great detail in the next chapter.

Review (TL;DR)

When attempting a property access on an object that doesn't have that property, the object's internal `[[Prototype]]` linkage defines where the `[[Get]]` operation (see Chapter 3) should look next. This cascading linkage from object to object essentially defines a "prototype chain" (somewhat similar to a nested scope chain) of objects to traverse for property resolution.

All normal objects have the built-in `Object.prototype` as the top of the prototype chain (like the global scope in scope look-up), where property resolution will stop if not found anywhere prior in the chain. `toString()`, `valueOf()`, and several other common utilities exist on this `Object.prototype` object, explaining how all objects in the language are able to access them.

The most common way to get two objects linked to each other is using the `new` keyword with a function call, which among its four steps (see Chapter 2), it creates a new object linked to another object.

The "another object" that the new object is linked to happens to be the object referenced by the arbitrarily named `.prototype` property of the function called with `new`. Functions called with `new` are often called "constructors", despite the fact that they are not actually instantiating a class as *constructors* do in traditional class-oriented languages.

While these JavaScript mechanisms can seem to resemble "class instantiation" and "class inheritance" from traditional class-oriented languages, the key distinction is that in JavaScript, no copies are made. Rather, objects end up linked to each other via an internal `[[Prototype]]` chain.

For a variety of reasons, not the least of which is terminology precedent, "inheritance" (and "prototypal inheritance") and all the other OO terms just do not make sense when considering how JavaScript *actually* works (not just applied to our forced mental models).

Instead, "delegation" is a more appropriate term, because these relationships are not *copies* but delegation **links**.

You Don't Know JS: *this* & Object Prototypes

Chapter 6: Behavior Delegation

In Chapter 5, we addressed the `[[Prototype]]` mechanism in detail, and *why* it's confusing and inappropriate (despite countless attempts for nearly two decades) to describe it as "class" or "inheritance". We trudged through not only the fairly verbose syntax (`.prototype` littering the code), but the various gotchas (like surprising `.constructor` resolution or ugly pseudo-polymorphic syntax). We explored variations of the "mixin" approach, which many people use to attempt to smooth over such rough areas.

It's a common reaction at this point to wonder why it has to be so complex to do something seemingly so simple. Now that we've pulled back the curtain and seen just how dirty it all gets, it's not a surprise that most JS developers never dive this deep, and instead relegate such mess to a "class" library to handle it for them.

I hope by now you're not content to just gloss over and leave such details to a "black box" library. Let's now dig into how we *could and should be* thinking about the object `[[Prototype]]` mechanism in JS, in a **much simpler and more straightforward way** than the confusion of classes.

As a brief review of our conclusions from Chapter 5, the `[[Prototype]]` mechanism is an internal link that exists on one object which references another object.

This linkage is exercised when a property/method reference is made against the first object, and no such property/method exists. In that case, the `[[Prototype]]` linkage tells the engine to look for the property/method on the linked-to object. In turn, if that object cannot fulfill the look-up, its `[[Prototype]]` is followed, and so on. This series of links between objects forms what is called the "prototype chain".

In other words, the actual mechanism, the essence of what's important to the functionality we can leverage in JavaScript, is **all about objects being linked to other objects.**

That single observation is fundamental and critical to understanding the motivations and approaches for the rest of this chapter!

Towards Delegation-Oriented Design

To properly focus our thoughts on how to use `[[Prototype]]` in the most straightforward way, we must recognize that it represents a fundamentally different design pattern from classes (see Chapter 4).

Note: *Some* principles of class-oriented design are still very valid, so don't toss out everything you know (just most of it!). For example, *encapsulation* is quite powerful, and is compatible (though not as common) with delegation.

We need to try to change our thinking from the class/inheritance design pattern to the behavior delegation design pattern. If you have done most or all of your programming in your education/career thinking in classes, this may be uncomfortable or feel unnatural. You may need to try this mental exercise quite a few times to get the hang of this very different way of thinking.

I'm going to walk you through some theoretical exercises first, then we'll look side-by-side at a more concrete example to give you practical context for your own code.

Class Theory

Let's say we have several similar tasks ("XYZ", "ABC", etc) that we need to model in our software.

With classes, the way you design the scenario is: define a general parent (base) class like `Task`, defining shared behavior for all the "alike" tasks. Then, you define child classes `XYZ` and `ABC`, both of which inherit from `Task`, and each of which adds specialized behavior to handle their respective tasks.

Importantly, the class design pattern will encourage you that to get the most out of inheritance, you will want to employ method overriding (and polymorphism), where you override the definition of some general `Task` method in your `XYZ` task, perhaps even making use of `super` to call to the base version of that method while adding more behavior to it. **You'll likely find quite a few places** where you can "abstract" out general behavior to the parent class and specialize (override) it in your child classes.

Here's some loose pseudo-code for that scenario:

```
class Task {
    id;

    // constructor `Task()`
    Task(ID) { id = ID; }
    outputTask() { output( id ); }
}

class XYZ inherits Task {
    label;

    // constructor `XYZ()`
    XYZ(ID,Label) { super( ID ); label = Label; }
    outputTask() { super(); output( label ); }
}

class ABC inherits Task {
    // ...
}
```

Now, you can instantiate one or more **copies** of the `XYZ` child class, and use those instance(s) to perform task "XYZ". These instances have **copies both** of the general `Task` defined behavior as well as the specific `XYZ` defined behavior. Likewise, instances of the `ABC` class would have copies of the `Task` behavior and the specific `ABC` behavior. After construction, you will generally only interact with these instances (and not the classes), as the instances each have copies of all the behavior you need to do the intended task.

Delegation Theory

But now let's try to think about the same problem domain, but using *behavior delegation* instead of *classes*.

You will first define an **object** (not a class, nor a `function` as most JS'rs would lead you to believe) called `Task`, and it will have concrete behavior on it that includes utility methods that various tasks can use (read: *delegate to*!). Then, for each task ("XYZ", "ABC"), you define an **object** to hold that task-specific data/behavior. You **link** your task-specific object(s) to the `Task` utility object, allowing them to delegate to it when they need to.

Basically, you think about performing task "XYZ" as needing behaviors from two sibling/peer objects (`XYZ` and `Task`) to accomplish it. But rather than needing to compose them together, via class copies, we can keep them in their separate objects, and we can allow `XYZ` object to **delegate to** `Task` when needed.

Here's some simple code to suggest how you accomplish that:

```
var Task = {
    setID: function(ID) { this.id = ID; },
    outputID: function() { console.log( this.id ); }
};

// make `XYZ` delegate to `Task`
var XYZ = Object.create( Task );

XYZ.prepareTask = function(ID,Label) {
    this.setID( ID );
    this.label = Label;
};

XYZ.outputTaskDetails = function() {
    this.outputID();
    console.log( this.label );
};

// ABC = Object.create( Task );
// ABC ... = ...
```

In this code, `Task` and `XYZ` are not classes (or functions), they're **just objects**. `XYZ` is set up via `Object.create(..)` to `[[Prototype]]` delegate to the `Task` object (see Chapter 5).

As compared to class-orientation (aka, OO -- object-oriented), I call this style of code **"OLOO"** (objects-linked-to-other-objects). All we *really* care about is that the `XYZ` object delegates to the `Task` object (as does the `ABC` object).

In JavaScript, the `[[Prototype]]` mechanism links **objects** to other **objects**. There are no abstract mechanisms like "classes", no matter how much you try to convince yourself otherwise. It's like paddling a canoe upstream: you *can* do it, but you're *choosing* to go against the natural current, so it's obviously **going to be harder to get where you're going.**

Some other differences to note with **OLOO style code**:

1. Both `id` and `label` data members from the previous class example are data properties directly on `XYZ` (neither is on `Task`). In general, with `[[Prototype]]` delegation involved, **you want state to be on the delegators** (`XYZ` , `ABC`), not on the delegate (`Task`).
2. With the class design pattern, we intentionally named `outputTask` the same on both parent (`Task`) and child (`XYZ`), so that we could take advantage of overriding (polymorphism). In behavior delegation, we do the opposite: **we avoid if at all possible naming things the same** at different levels of the `[[Prototype]]` chain (called shadowing -- see Chapter 5), because having those name collisions creates awkward/brittle syntax to disambiguate references (see Chapter 4), and we want to avoid that if we can.

This design pattern calls for less of general method names which are prone to overriding and instead more of descriptive method names, *specific* to the type of behavior each object is doing. **This can actually create easier to understand/maintain code**, because the names of methods (not only at definition location but strewn throughout other code) are more obvious (self documenting).

3. `this.setID(ID);` inside of a method on the `XYZ` object first looks on `XYZ` for `setID(..)`, but since it doesn't find a method of that name on `XYZ`, `[[Prototype]]` *delegation* means it can follow the link to `Task` to look for `setID(..)`, which it of course finds. Moreover, because of implicit call-site `this` binding rules (see Chapter 2), when `setID(..)` runs, even though the method was found on `Task`, the `this` binding for that function call is `XYZ` exactly as we'd expect and want. We see the same thing with `this.outputID()` later in the code listing.

   In other words, the general utility methods that exist on `Task` are available to us while interacting with `XYZ`, because `XYZ` can delegate to `Task`.

Behavior Delegation means: let some object (`XYZ`) provide a delegation (to `Task`) for property or method references if not found on the object (`XYZ`).

This is an *extremely powerful* design pattern, very distinct from the idea of parent and child classes, inheritance, polymorphism, etc. Rather than organizing the objects in your mind vertically, with Parents flowing down to Children, think of objects side-by-side, as peers, with any direction of delegation links between the objects as necessary.

Note: Delegation is more properly used as an internal implementation detail rather than exposed directly in the API interface design. In the above example, we don't necessarily *intend* with our API design for developers to call `XYZ.setID()` (though we can, of course!). We sorta *hide* the delegation as an internal detail of our API, where `XYZ.prepareTask(..)` delegates to `Task.setID(..)`. See the "Links As Fallbacks?" discussion in Chapter 5 for more detail.

Mutual Delegation (Disallowed)

You cannot create a *cycle* where two or more objects are mutually delegated (bi-directionally) to each other. If you make `B` linked to `A`, and then try to link `A` to `B`, you will get an error.

It's a shame (not terribly surprising, but mildly annoying) that this is disallowed. If you made a reference to a property/method which didn't exist in either place, you'd have an infinite recursion on the `[[Prototype]]` loop. But if all references were strictly present, then `B`

could delegate to `A`, and vice versa, and it *could* work. This would mean you could use either object to delegate to the other, for various tasks. There are a few niche use-cases where this might be helpful.

But it's disallowed because engine implementors have observed that it's more performant to check for (and reject!) the infinite circular reference once at set-time rather than needing to have the performance hit of that guard check every time you look-up a property on an object.

Debugged

We'll briefly cover a subtle detail that can be confusing to developers. In general, the JS specification does not control how browser developer tools should represent specific values/structures to a developer, so each browser/engine is free to interpret such things as they see fit. As such, browsers/tools *don't always agree*. Specifically, the behavior we will now examine is currently observed only in Chrome's Developer Tools.

Consider this traditional "class constructor" style JS code, as it would appear in the *console* of Chrome Developer Tools:

```
function Foo() {}

var a1 = new Foo();

a1; // Foo {}
```

Let's look at the last line of that snippet: the output of evaluating the `a1` expression, which prints `Foo {}`. If you try this same code in Firefox, you will likely see `Object {}`. Why the difference? What do these outputs mean?

Chrome is essentially saying "{} is an empty object that was constructed by a function with name 'Foo'". Firefox is saying "{} is an empty object of general construction from Object". The subtle difference is that Chrome is actively tracking, as an *internal property*, the name of the actual function that did the construction, whereas other browsers don't track that additional information.

It would be tempting to attempt to explain this with JavaScript mechanisms:

```
function Foo() {}

var a1 = new Foo();

a1.constructor; // Foo(){}
a1.constructor.name; // "Foo"
```

So, is that how Chrome is outputting "Foo", by simply examining the object's `.constructor.name`? Confusingly, the answer is both "yes" and "no".

Consider this code:

```
function Foo() {}

var a1 = new Foo();

Foo.prototype.constructor = function Gotcha(){};

a1.constructor; // Gotcha(){}
a1.constructor.name; // "Gotcha"

a1; // Foo {}
```

Even though we change `a1.constructor.name` to legitimately be something else ("Gotcha"), Chrome's console still uses the "Foo" name.

So, it would appear the answer to previous question (does it use `.constructor.name`?) is **no**, it must track it somewhere else, internally.

But, Not so fast! Let's see how this kind of behavior works with OLOO-style code:

```
var Foo = {};

var a1 = Object.create( Foo );

a1; // Object {}

Object.defineProperty( Foo, "constructor", {
    enumerable: false,
    value: function Gotcha(){}
});

a1; // Gotcha {}
```

Ah-ha! **Gotcha!** Here, Chrome's console **did** find and use the `.constructor.name`. Actually, while writing this book, this exact behavior was identified as a bug in Chrome, and by the time you're reading this, it may have already been fixed. So you may instead have seen the corrected `a1; // Object {}`.

Aside from that bug, the internal tracking (apparently only for debug output purposes) of the "constructor name" that Chrome does (shown in the earlier snippets) is an intentional Chrome-only extension of behavior beyond what the JS specification calls for.

If you don't use a "constructor" to make your objects, as we've discouraged with OLOO-style code here in this chapter, then you'll get objects that Chrome does *not* track an internal "constructor name" for, and such objects will correctly only be outputted as "Object {}", meaning "object generated from Object() construction".

Don't think this represents a drawback of OLOO-style coding. When you code with OLOO and behavior delegation as your design pattern, *who* "constructed" (that is, *which function* was called with `new`?) some object is an irrelevant detail. Chrome's specific internal "constructor name" tracking is really only useful if you're fully embracing "class-style" coding, but is moot if you're instead embracing OLOO delegation.

Mental Models Compared

Now that you can see a difference between "class" and "delegation" design patterns, at least theoretically, let's see the implications these design patterns have on the mental models we use to reason about our code.

We'll examine some more theoretical ("Foo", "Bar") code, and compare both ways (OO vs. OLOO) of implementing the code. The first snippet uses the classical ("prototypal") OO style:

```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

function Bar(who) {
    Foo.call( this, who );
}
Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = new Bar( "b1" );
var b2 = new Bar( "b2" );

b1.speak();
b2.speak();
```

Parent class `Foo`, inherited by child class `Bar`, which is then instantiated twice as `b1` and `b2`. What we have is `b1` delegating to `Bar.prototype` which delegates to `Foo.prototype`. This should look fairly familiar to you, at this point. Nothing too ground-breaking going on.

Now, let's implement **the exact same functionality** using *OLOO* style code:

```
var Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};

var Bar = Object.create( Foo );

Bar.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = Object.create( Bar );
b1.init( "b1" );
var b2 = Object.create( Bar );
b2.init( "b2" );

b1.speak();
b2.speak();
```

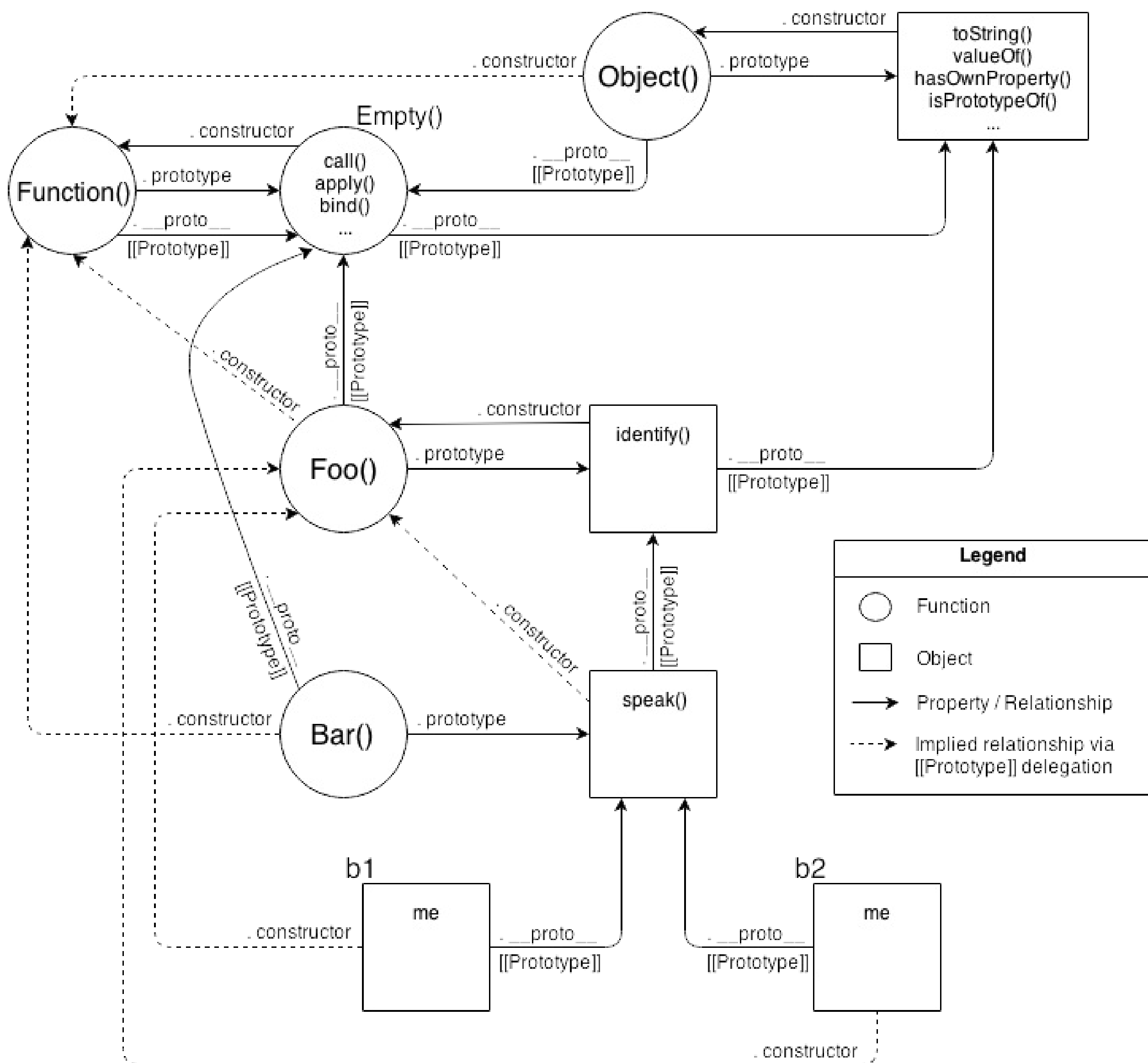
We take exactly the same advantage of `[[Prototype]]` delegation from `b1` to `Bar` to `Foo` as we did in the previous snippet between `b1`, `Bar.prototype`, and `Foo.prototype`. **We still have the same 3 objects linked together**.

But, importantly, we've greatly simplified *all the other stuff* going on, because now we just set up **objects** linked to each other, without needing all the cruft and confusion of things that look (but don't behave!) like classes, with constructors and prototypes and `new` calls.

Ask yourself: if I can get the same functionality with OLOO style code as I do with "class" style code, but OLOO is simpler and has less things to think about, **isn't OLOO better**?

Let's examine the mental models involved between these two snippets.

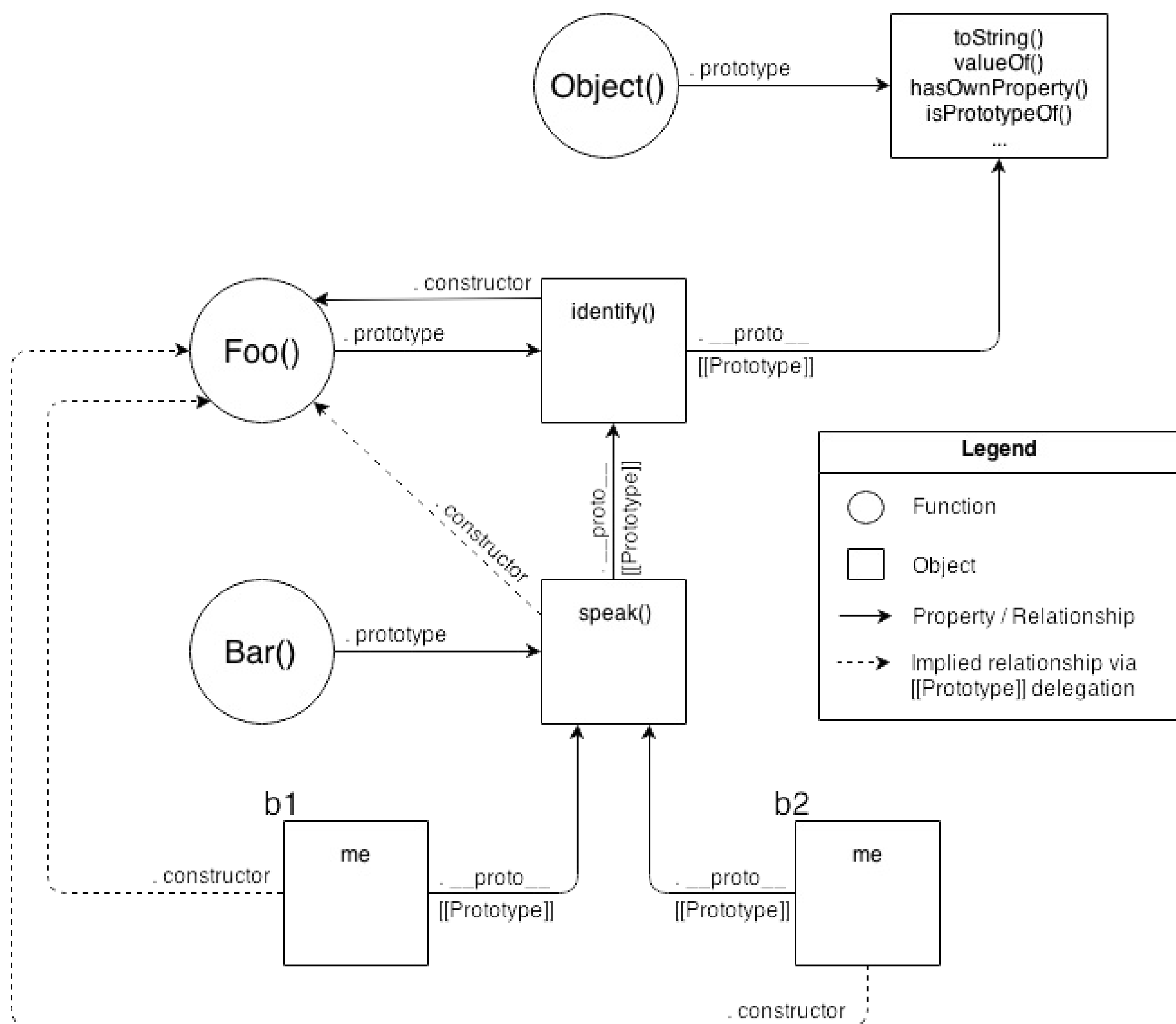
First, the class-style code snippet implies this mental model of entities and their relationships:



Actually, that's a little unfair/misleading, because it's showing a lot of extra detail that you don't *technically* need to know at all times (though you *do* need to understand it!). One take-away is that it's quite a complex series of relationships. But another take-away: if you spend the time to follow those relationship arrows around, **there's an amazing amount of internal consistency** in JS's mechanisms.

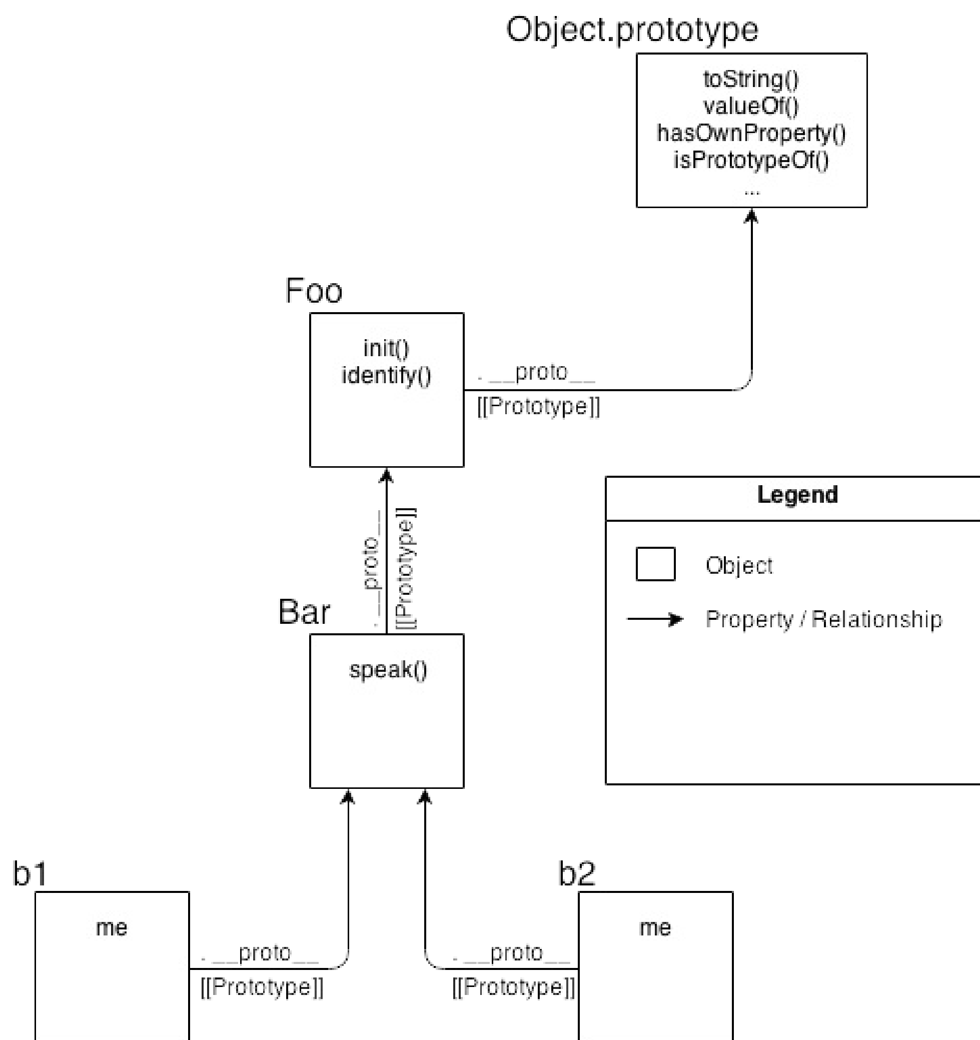
For instance, the ability of a JS function to access `call(..)`, `apply(..)`, and `bind(..)` (see Chapter 2) is because functions themselves are objects, and function-objects also have a `[[Prototype]]` linkage, to the `Function.prototype` object, which defines those default methods that any function-object can delegate to. JS can do those things, *and you can too!*.

OK, let's now look at a *slightly* simplified version of that diagram which is a little more "fair" for comparison -- it shows only the *relevant* entities and relationships.



Still pretty complex, eh? The dotted lines are depicting the implied relationships when you setup the "inheritance" between `Foo.prototype` and `Bar.prototype` and haven't yet *fixed* the **missing** `.constructor` property reference (see "Constructor Redux" in Chapter 5). Even with those dotted lines removed, the mental model is still an awful lot to juggle every time you work with object linkages.

Now, let's look at the mental model for OLOO-style code:



As you can see comparing them, it's quite obvious that OLOO-style code has *vastly less stuff* to worry about, because OLOO-style code embraces the **fact** that the only thing we ever really cared about was the **objects linked to other objects**.

All the other "class" cruft was a confusing and complex way of getting the same end result. Remove that stuff, and things get much simpler (without losing any capability).

Classes vs. Objects

We've just seen various theoretical explorations and mental models of "classes" vs. "behavior delegation". But, let's now look at more concrete code scenarios to show how'd you actually use these ideas.

We'll first examine a typical scenario in front-end web dev: creating UI widgets (buttons, drop-downs, etc).

Widget "Classes"

Because you're probably still so used to the OO design pattern, you'll likely immediately think of this problem domain in terms of a parent class (perhaps called `Widget`) with all the common base widget behavior, and then child derived classes for specific widget types (like `Button`).

Note: We're going to use jQuery here for DOM and CSS manipulation, only because it's a detail we don't really care about for the purposes of our current discussion. None of this code cares which JS framework (jQuery, Dojo, YUI, etc), if any, you might solve such mundane tasks with.

Let's examine how we'd implement the "class" design in classic-style pure JS without any "class" helper library or syntax:

```
// Parent class
function Widget(width,height) {
    this.width = width || 50;
    this.height = height || 50;
    this.$elem = null;
}

Widget.prototype.render = function($where){
    if (this.$elem) {
        this.$elem.css( {
            width: this.width + "px",
            height: this.height + "px"
        } ).appendTo( $where );
    }
};

// Child class
function Button(width,height,label) {
    // "super" constructor call
    Widget.call( this, width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
}

// make `Button` "inherit" from `Widget`
Button.prototype = Object.create( Widget.prototype );

// override base "inherited" `render(..)`
Button.prototype.render = function($where) {
    // "super" call
    Widget.prototype.render.call( this, $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.prototype.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );
```

OO design patterns tell us to declare a base `render(..)` in the parent class, then override it in our child class, but not to replace it per se, rather to augment the base functionality with button-specific behavior.

Notice the ugliness of *explicit pseudo-polymorphism* (see Chapter 4) with `Widget.call` and `Widget.prototype.render.call` references for faking "super" calls from the child "class" methods back up to the "parent" class base methods. Yuck.

ES6 `class` sugar

We cover ES6 `class` syntax sugar in detail in Appendix A, but let's briefly demonstrate how we'd implement the same code using `class` :

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );
```

Undoubtedly, a number of the syntax uglies of the previous classical approach have been smoothed over with ES6's `class`. The presence of a `super(..)` in particular seems quite nice (though when you dig into it, it's not all roses!).

Despite syntactic improvements, **these are not *real* classes**, as they still operate on top of the `[[Prototype]]` mechanism. They suffer from all the same mental-model mismatches we explored in Chapters 4, 5 and thus far in this chapter. Appendix A will expound on the ES6 `class` syntax and its implications in detail. We'll see why solving syntax hiccups doesn't substantially solve our class confusions in JS, though it makes a valiant effort masquerading as a solution!

Whether you use the classic prototypal syntax or the new ES6 sugar, you've still made a *choice* to model the problem domain (UI widgets) with "classes". And as the previous few chapters try to demonstrate, this *choice* in JavaScript is opting you into extra headaches and mental tax.

Delegating Widget Objects

Here's our simpler `Widget` / `Button` example, using **OLOO style delegation**:

```
var Widget = {
    init: function(width,height){
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    },
    insert: function($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
};

var Button = Object.create( Widget );

Button.setup = function(width,height,label){
    // delegated call
    this.init( width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
};
Button.build = function($where) {
    // delegated call
    this.insert( $where );
    this.$elem.click( this.onClick.bind( this ) );
};
Button.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );

    var btn1 = Object.create( Button );
    btn1.setup( 125, 30, "Hello" );

    var btn2 = Object.create( Button );
    btn2.setup( 150, 40, "World" );

    btn1.build( $body );
    btn2.build( $body );
} );
```

With this OLOO-style approach, we don't think of `Widget` as a parent and `Button` as a child. Rather, `Widget` **is just an object** and is sort of a utility collection that any specific type of widget might want to delegate to, and `Button` **is also just a stand-alone object**

(with a delegation link to `Widget`, of course!).

From a design pattern perspective, we **didn't** share the same method name `render(..)` in both objects, the way classes suggest, but instead we chose different names (`insert(..)` and `build(..)`) that were more descriptive of what task each does specifically. The *initialization* methods are called `init(..)` and `setup(..)`, respectively, for the same reasons.

Not only does this delegation design pattern suggest different and more descriptive names (rather than shared and more generic names), but doing so with OLOO happens to avoid the ugliness of the explicit pseudo-polymorphic calls (`Widget.call` and `Widget.prototype.render.call`), as you can see by the simple, relative, delegated calls to `this.init(..)` and `this.insert(..)`.

Syntactically, we also don't have any constructors, `.prototype` or `new` present, as they are, in fact, just unnecessary cruft.

Now, if you're paying close attention, you may notice that what was previously just one call (`var btn1 = new Button(..)`) is now two calls (`var btn1 = Object.create(Button)` and `btn1.setup(..)`). Initially this may seem like a drawback (more code).

However, even this is something that's **a pro of OLOO style code** as compared to classical prototype style code. How?

With class constructors, you are "forced" (not really, but strongly suggested) to do both construction and initialization in the same step. However, there are many cases where being able to do these two steps separately (as you do with OLOO!) is more flexible.

For example, let's say you create all your instances in a pool at the beginning of your program, but you wait to initialize them with specific setup until they are pulled from the pool and used. We showed the two calls happening right next to each other, but of course they can happen at very different times and in very different parts of our code, as needed.

OLOO supports *better* the principle of separation of concerns, where creation and initialization are not necessarily conflated into the same operation.

Simpler Design

In addition to OLOO providing ostensibly simpler (and more flexible!) code, behavior delegation as a pattern can actually lead to simpler code architecture. Let's examine one last example that illustrates how OLOO simplifies your overall design.

The scenario we'll examine is two controller objects, one for handling the login form of a web page, and another for actually handling the authentication (communication) with the server.

We'll need a utility helper for making the Ajax communication to the server. We'll use jQuery (though any framework would do fine), since it handles not only the Ajax for us, but it returns a promise-like answer so that we can listen for the response in our calling code with `.then(..)`.

Note: We don't cover Promises here, but we will cover them in a future title of the *"You Don't Know JS"* series.

Following the typical class design pattern, we'll break up the task into base functionality in a class called `Controller`, and then we'll derive two child classes, `LoginController` and `AuthController`, which both inherit from `Controller` and specialize some of those base behaviors.

```
// Parent class
function Controller() {
    this.errors = [];
}
Controller.prototype.showDialog = function(title,msg) {
    // display title & message to user in dialog
};
Controller.prototype.success = function(msg) {
    this.showDialog( "Success", msg );
};
Controller.prototype.failure = function(err) {
    this.errors.push( err );
    this.showDialog( "Error", err );
};
```

```
// Child class
function LoginController() {
    Controller.call( this );
}
// Link child class to parent
LoginController.prototype = Object.create( Controller.prototype );
LoginController.prototype.getUser = function() {
    return document.getElementById( "login_username" ).value;
};
LoginController.prototype.getPassword = function() {
    return document.getElementById( "login_password" ).value;
};
LoginController.prototype.validateEntry = function(user,pw) {
    user = user || this.getUser();
    pw = pw || this.getPassword();

    if (!(user && pw)) {
        return this.failure( "Please enter a username & password!" );
    }
    else if (pw.length < 5) {
        return this.failure( "Password must be 5+ characters!" );
    }

    // got here? validated!
    return true;
};
// Override to extend base `failure()`
LoginController.prototype.failure = function(err) {
    // "super" call
    Controller.prototype.failure.call( this, "Login invalid: " + err );
};
```

```js
// Child class
function AuthController(login) {
    Controller.call( this );
    // in addition to inheritance, we also need composition
    this.login = login;
}
// Link child class to parent
AuthController.prototype = Object.create( Controller.prototype );
AuthController.prototype.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};
AuthController.prototype.checkAuth = function() {
    var user = this.login.getUser();
    var pw = this.login.getPassword();

    if (this.login.validateEntry( user, pw )) {
        this.server( "/check-auth",{
            user: user,
            pw: pw
        } )
        .then( this.success.bind( this ) )
        .fail( this.failure.bind( this ) );
    }
};
// Override to extend base `success()`
AuthController.prototype.success = function() {
    // "super" call
    Controller.prototype.success.call( this, "Authenticated!" );
};
// Override to extend base `failure()`
AuthController.prototype.failure = function(err) {
    // "super" call
    Controller.prototype.failure.call( this, "Auth Failed: " + err );
};
```

```js
var auth = new AuthController(
    // in addition to inheritance, we also need composition
    new LoginController()
);
auth.checkAuth();
```

We have base behaviors that all controllers share, which are `success(..)`, `failure(..)` and `showDialog(..)`. Our child classes `LoginController` and `AuthController` override `failure(..)` and `success(..)` to augment the default base class behavior. Also note that

`AuthController` needs an instance of `LoginController` to interact with the login form, so that becomes a member data property.

The other thing to mention is that we chose some *composition* to sprinkle in on top of the inheritance. `AuthController` needs to know about `LoginController`, so we instantiate it (`new LoginController()`) and keep a class member property called `this.login` to reference it, so that `AuthController` can invoke behavior on `LoginController`.

Note: There *might* have been a slight temptation to make `AuthController` inherit from `LoginController`, or vice versa, such that we had *virtual composition* through the inheritance chain. But this is a strongly clear example of what's wrong with class inheritance as *the* model for the problem domain, because neither `AuthController` nor `LoginController` are specializing base behavior of the other, so inheritance between them makes little sense except if classes are your only design pattern. Instead, we layered in some simple *composition* and now they can cooperate, while still both benefiting from the inheritance from the parent base `Controller`.

If you're familiar with class-oriented (OO) design, this should all look pretty familiar and natural.

De-class-ified

But, **do we really need to model this problem** with a parent `Controller` class, two child classes, **and some composition**? Is there a way to take advantage of OLOO-style behavior delegation and have a *much* simpler design? **Yes!**

```
var LoginController = {
    errors: [],
    getUser: function() {
        return document.getElementById( "login_username" ).value;
    },
    getPassword: function() {
        return document.getElementById( "login_password" ).value;
    },
    validateEntry: function(user,pw) {
        user = user || this.getUser();
        pw = pw || this.getPassword();

        if (!(user && pw)) {
            return this.failure( "Please enter a username & password!" );
        }
        else if (pw.length < 5) {
            return this.failure( "Password must be 5+ characters!" );
        }

        // got here? validated!
        return true;
    },
    showDialog: function(title,msg) {
        // display success message to user in dialog
    },
    failure: function(err) {
        this.errors.push( err );
        this.showDialog( "Error", "Login invalid: " + err );
    }
};
```

```
// Link `AuthController` to delegate to `LoginController`
var AuthController = Object.create( LoginController );

AuthController.errors = [];
AuthController.checkAuth = function() {
    var user = this.getUser();
    var pw = this.getPassword();

    if (this.validateEntry( user, pw )) {
        this.server( "/check-auth",{
            user: user,
            pw: pw
        } )
        .then( this.accepted.bind( this ) )
        .fail( this.rejected.bind( this ) );
    }
};
AuthController.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};
AuthController.accepted = function() {
    this.showDialog( "Success", "Authenticated!" )
};
AuthController.rejected = function(err) {
    this.failure( "Auth Failed: " + err );
};
```

Since `AuthController` is just an object (so is `LoginController`), we don't need to instantiate (like `new AuthController()`) to perform our task. All we need to do is:

```
AuthController.checkAuth();
```

Of course, with OLOO, if you do need to create one or more additional objects in the delegation chain, that's easy, and still doesn't require anything like class instantiation:

```
var controller1 = Object.create( AuthController );
var controller2 = Object.create( AuthController );
```

With behavior delegation, `AuthController` and `LoginController` are **just objects**, *horizontal* peers of each other, and are not arranged or related as parents and children in class-orientation. We somewhat arbitrarily chose to have `AuthController` delegate to `LoginController` -- it would have been just as valid for the delegation to go the reverse direction.

The main takeaway from this second code listing is that we only have two entities (`LoginController` and `AuthController`), **not three** as before.

We didn't need a base `Controller` class to "share" behavior between the two, because delegation is a powerful enough mechanism to give us the functionality we need. We also, as noted before, don't need to instantiate our classes to work with them, because there are no classes, **just the objects themselves.** Furthermore, there's no need for *composition* as delegation gives the two objects the ability to cooperate *differentially* as needed.

Lastly, we avoided the polymorphism pitfalls of class-oriented design by not having the names `success(..)` and `failure(..)` be the same on both objects, which would have required ugly explicit pseudopolymorphism. Instead, we called them `accepted()` and `rejected(..)` on `AuthController` -- slightly more descriptive names for their specific tasks.

Bottom line: we end up with the same capability, but a (significantly) simpler design. That's the power of OLOO-style code and the power of the *behavior delegation* design pattern.

Nicer Syntax

One of the nicer things that makes ES6's `class` so deceptively attractive (see Appendix A on why to avoid it!) is the short-hand syntax for declaring class methods:

```
class Foo {
    methodName() { /* .. */ }
}
```

We get to drop the word `function` from the declaration, which makes JS developers everywhere cheer!

And you may have noticed and been frustrated that the suggested OLOO syntax above has lots of `function` appearances, which seems like a bit of a detractor to the goal of OLOO simplification. **But it doesn't have to be that way!**

As of ES6, we can use *concise method declarations* in any object literal, so an object in OLOO style can be declared this way (same short-hand sugar as with `class` body syntax):

```
var LoginController = {
    errors: [],
    getUser() { // Look ma, no `function`!
        // ...
    },
    getPassword() {
        // ...
    }
    // ...
};
```

About the only difference is that object literals will still require `,` comma separators between elements whereas `class` syntax doesn't. Pretty minor concession in the whole scheme of things.

Moreover, as of ES6, the clunkier syntax you use (like for the `AuthController` definition), where you're assigning properties individually and not using an object literal, can be re-written using an object literal (so that you can use concise methods), and you can just modify that object's `[[Prototype]]` with `Object.setPrototypeOf(..)`, like this:

```
// use nicer object literal syntax w/ concise methods!
var AuthController = {
    errors: [],
    checkAuth() {
        // ...
    },
    server(url,data) {
        // ...
    }
    // ...
};

// NOW, link `AuthController` to delegate to `LoginController`
Object.setPrototypeOf( AuthController, LoginController );
```

OLOO-style as of ES6, with concise methods, **is a lot friendlier** than it was before (and even then, it was much simpler and nicer than classical prototype-style code). **You don't have to opt for class** (complexity) to get nice clean object syntax!

Unlexical

There *is* one drawback to concise methods that's subtle but important to note. Consider this code:

```
var Foo = {
    bar() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

Here's the syntactic de-sugaring that expresses how that code will operate:

```
var Foo = {
    bar: function() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

See the difference? The `bar()` short-hand became an *anonymous function expression* (`function()..`) attached to the `bar` property, because the function object itself has no name identifier. Compare that to the manually specified *named function expression* (`function baz()..`) which has a lexical name identifier `baz` in addition to being attached to a `.baz` property.

So what? In the *"Scope & Closures"* title of this *"You Don't Know JS"* book series, we cover the three main downsides of *anonymous function expressions* in detail. We'll just briefly repeat them so we can compare to the concise method short-hand.

Lack of a `name` identifier on an anonymous function:

1. makes debugging stack traces harder
2. makes self-referencing (recursion, event (un)binding, etc) harder
3. makes code (a little bit) harder to understand

Items 1 and 3 don't apply to concise methods.

Even though the de-sugaring uses an *anonymous function expression* which normally would have no `name` in stack traces, concise methods are specified to set the internal `name` property of the function object accordingly, so stack traces should be able to use it (though that's implementation dependent so not guaranteed).

Item 2 is, unfortunately, **still a drawback to concise methods**. They will not have a lexical identifier to use as a self-reference. Consider:

```
var Foo = {
    bar: function(x) {
        if (x < 10) {
            return Foo.bar( x * 2 );
        }
        return x;
    },
    baz: function baz(x) {
        if (x < 10) {
            return baz( x * 2 );
        }
        return x;
    }
};
```

The manual `Foo.bar(x*2)` reference above kind of suffices in this example, but there are many cases where a function wouldn't necessarily be able to do that, such as cases where the function is being shared in delegation across different objects, using `this` binding, etc. You would want to use a real self-reference, and the function object's `name` identifier is the best way to accomplish that.

Just be aware of this caveat for concise methods, and if you run into such issues with lack of self-reference, make sure to forgo the concise method syntax **just for that declaration** in favor of the manual *named function expression* declaration form: `baz: function baz(){..}` .

Introspection

If you've spent much time with class oriented programming (either in JS or other languages), you're probably familiar with *type introspection*: inspecting an instance to find out what *kind* of object it is. The primary goal of *type introspection* with class instances is to reason about the structure/capabilities of the object based on *how it was created*.

Consider this code which uses `instanceof` (see Chapter 5) for introspecting on an object `a1` to infer its capability:

```
function Foo() {
    // ...
}
Foo.prototype.something = function(){
    // ...
}

var a1 = new Foo();

// later

if (a1 instanceof Foo) {
    a1.something();
}
```

Because `Foo.prototype` (not `Foo`!) is in the `[[Prototype]]` chain (see Chapter 5) of `a1`, the `instanceof` operator (confusingly) pretends to tell us that `a1` is an instance of the `Foo` "class". With this knowledge, we then assume that `a1` has the capabilities described by the `Foo` "class".

Of course, there is no `Foo` class, only a plain old normal function `Foo`, which happens to have a reference to an arbitrary object (`Foo.prototype`) that `a1` happens to be delegation-linked to. By its syntax, `instanceof` pretends to be inspecting the relationship between `a1` and `Foo`, but it's actually telling us whether `a1` and (the arbitrary object referenced by) `Foo.prototype` are related.

The semantic confusion (and indirection) of `instanceof` syntax means that to use `instanceof`-based introspection to ask if object `a1` is related to the capabilities object in question, you *have to* have a function that holds a reference to that object -- you can't just directly ask if the two objects are related.

Recall the abstract `Foo` / `Bar` / `b1` example from earlier in this chapter, which we'll abbreviate here:

```
function Foo() { /* .. */ }
Foo.prototype...

function Bar() { /* .. */ }
Bar.prototype = Object.create( Foo.prototype );

var b1 = new Bar( "b1" );
```

For *type introspection* purposes on the entities in that example, using `instanceof` and `.prototype` semantics, here are the various checks you might need to perform:

```
// relating `Foo` and `Bar` to each other
Bar.prototype instanceof Foo; // true
Object.getPrototypeOf( Bar.prototype ) === Foo.prototype; // true
Foo.prototype.isPrototypeOf( Bar.prototype ); // true

// relating `b1` to both `Foo` and `Bar`
b1 instanceof Foo; // true
b1 instanceof Bar; // true
Object.getPrototypeOf( b1 ) === Bar.prototype; // true
Foo.prototype.isPrototypeOf( b1 ); // true
Bar.prototype.isPrototypeOf( b1 ); // true
```

It's fair to say that some of that kinda sucks. For instance, intuitively (with classes) you might want to be able to say something like `Bar instanceof Foo` (because it's easy to mix up what "instance" means to think it includes "inheritance"), but that's not a sensible comparison in JS. You have to do `Bar.prototype instanceof Foo` instead.

Another common, but perhaps less robust, pattern for *type introspection*, which many devs seem to prefer over `instanceof`, is called "duck typing". This term comes from the adage, "if it looks like a duck, and it quacks like a duck, it must be a duck".

Example:

```
if (a1.something) {
    a1.something();
}
```

Rather than inspecting for a relationship between `a1` and an object that holds the delegatable `something()` function, we assume that the test for `a1.something` passing means `a1` has the capability to call `.something()` (regardless of if it found the method directly on `a1` or delegated to some other object). In and of itself, that assumption isn't so risky.

But "duck typing" is often extended to make **other assumptions about the object's capabilities** besides what's being tested, which of course introduces more risk (aka, brittle design) into the test.

One notable example of "duck typing" comes with ES6 Promises (which as an earlier note explained are not being covered in this book).

For various reasons, there's a need to determine if any arbitrary object reference *is a Promise*, but the way that test is done is to check if the object happens to have a `then()` function present on it. In other words, **if any object** happens to have a `then()` method, ES6 Promises will assume unconditionally that the object **is a "thenable"** and therefore will expect it to behave conformantly to all standard behaviors of Promises.

If you have any non-Promise object that happens for whatever reason to have a `then()` method on it, you are strongly advised to keep it far away from the ES6 Promise mechanism to avoid broken assumptions.

That example clearly illustrates the perils of "duck typing". You should only use such approaches sparingly and in controlled conditions.

Turning our attention once again back to OLOO-style code as presented here in this chapter, *type introspection* turns out to be much cleaner. Let's recall (and abbreviate) the `Foo` / `Bar` / `b1` OLOO example from earlier in the chapter:

```
var Foo = { /* .. */ };

var Bar = Object.create( Foo );
Bar...

var b1 = Object.create( Bar );
```

Using this OLOO approach, where all we have are plain objects that are related via `[[Prototype]]` delegation, here's the quite simplified *type introspection* we might use:

```
// relating `Foo` and `Bar` to each other
Foo.isPrototypeOf( Bar ); // true
Object.getPrototypeOf( Bar ) === Foo; // true

// relating `b1` to both `Foo` and `Bar`
Foo.isPrototypeOf( b1 ); // true
Bar.isPrototypeOf( b1 ); // true
Object.getPrototypeOf( b1 ) === Bar; // true
```

We're not using `instanceof` anymore, because it's confusingly pretending to have something to do with classes. Now, we just ask the (informally stated) question, "are you *a* prototype of me?" There's no more indirection necessary with stuff like `Foo.prototype` or the painfully verbose `Foo.prototype.isPrototypeOf(..)`.

I think it's fair to say these checks are significantly less complicated/confusing than the previous set of introspection checks. **Yet again, we see that OLOO is simpler than (but with all the same power of) class-style coding in JavaScript.**

Review (TL;DR)

Classes and inheritance are a design pattern you can *choose*, or *not choose*, in your software architecture. Most developers take for granted that classes are the only (proper) way to organize code, but here we've seen there's another less-commonly talked about pattern that's actually quite powerful: **behavior delegation**.

Behavior delegation suggests objects as peers of each other, which delegate amongst themselves, rather than parent and child class relationships. JavaScript's `[[Prototype]]` mechanism is, by its very designed nature, a behavior delegation mechanism. That means we can either choose to struggle to implement class mechanics on top of JS (see Chapters 4 and 5), or we can just embrace the natural state of `[[Prototype]]` as a delegation mechanism.

When you design code with objects only, not only does it simplify the syntax you use, but it can actually lead to simpler code architecture design.

OLOO (objects-linked-to-other-objects) is a code style which creates and relates objects directly without the abstraction of classes. OLOO quite naturally implements `[[Prototype]]`-based behavior delegation.

You Don't Know JS: *this* & Object Prototypes

Appendix A: ES6 `class`

If there's any take-away message from the second half of this book (Chapters 4-6), it's that classes are an optional design pattern for code (not a necessary given), and that furthermore they are often quite awkward to implement in a `[[Prototype]]` language like JavaScript.

This awkwardness is *not* just about syntax, although that's a big part of it. Chapters 4 and 5 examined quite a bit of syntactic ugliness, from verbosity of `.prototype` references cluttering the code, to *explicit pseudo-polymorphism* (see Chapter 4) when you give methods the same name at different levels of the chain and try to implement a polymorphic reference from a lower-level method to a higher-level method. `.constructor` being wrongly interpreted as "was constructed by" and yet being unreliable for that definition is yet another syntactic ugly.

But the problems with class design are much deeper. Chapter 4 points out that classes in traditional class-oriented languages actually produce a *copy* action from parent to child to instance, whereas in `[[Prototype]]`, the action is **not** a copy, but rather the opposite -- a delegation link.

When compared to the simplicity of OLOO-style code and behavior delegation (see Chapter 6), which embrace `[[Prototype]]` rather than hide from it, classes stand out as a sore thumb in JS.

`class`

But we *don't* need to re-argue that case again. I re-mention those issues briefly only so that you keep them fresh in your mind now that we turn our attention to the ES6 `class` mechanism. We'll demonstrate here how it works, and look at whether or not `class` does anything substantial to address any of those "class" concerns.

Let's revisit the `Widget` / `Button` example from Chapter 6:

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}
```

Beyond this syntax *looking* nicer, what problems does ES6 solve?

1. There's no more (well, sorta, see below!) references to `.prototype` cluttering the code.
2. `Button` is declared directly to "inherit from" (aka `extends`) `Widget`, instead of needing to use `Object.create(..)` to replace a `.prototype` object that's linked, or having to set with `.__proto__` or `Object.setPrototypeOf(..)`.
3. `super(..)` now gives us a very helpful **relative polymorphism** capability, so that any method at one level of the chain can refer relatively one level up the chain to a method of the same name. This includes a solution to the note from Chapter 4 about the weirdness of constructors not belonging to their class, and so being unrelated -- `super()` works inside constructors exactly as you'd expect.
4. `class` literal syntax has no affordance for specifying properties (only methods). This might seem limiting to some, but it's expected that the vast majority of cases where a property (state) exists elsewhere but the end-chain "instances", this is usually a mistake and surprising (as it's state that's implicitly "shared" among all "instances"). So, one *could* say the `class` syntax is protecting you from mistakes.

5. `extends` lets you extend even built-in object (sub)types, like `Array` or `RegExp`, in a very natural way. Doing so without `class .. extends` has long been an exceedingly complex and frustrating task, one that only the most adept of framework authors have ever been able to accurately tackle. Now, it will be rather trivial!

In all fairness, those are some substantial solutions to many of the most obvious (syntactic) issues and surprises people have with classical prototype-style code.

`class` Gotchas

It's not all bubblegum and roses, though. There are still some deep and profoundly troubling issues with using "classes" as a design pattern in JS.

Firstly, the `class` syntax may convince you a new "class" mechanism exists in JS as of ES6. **Not so.** `class` is, mostly, just syntactic sugar on top of the existing `[[Prototype]]` (delegation!) mechanism.

That means `class` is not actually copying definitions statically at declaration time the way it does in traditional class-oriented languages. If you change/replace a method (on purpose or by accident) on the parent "class", the child "class" and/or instances will still be "affected", in that they didn't get copies at declaration time, they are all still using the live-delegation model based on `[[Prototype]]`:

```
class C {
    constructor() {
        this.num = Math.random();
    }
    rand() {
        console.log( "Random: " + this.num );
    }
}

var c1 = new C();
c1.rand(); // "Random: 0.4324299..."

C.prototype.rand = function() {
    console.log( "Random: " + Math.round( this.num * 1000 ));
};

var c2 = new C();
c2.rand(); // "Random: 867"

c1.rand(); // "Random: 432" -- oops!!!
```

This only seems like reasonable behavior *if you already know* about the delegation nature of things, rather than expecting *copies* from "real classes". So the question to ask yourself is, why are you choosing `class` syntax for something fundamentally different from classes?

Doesn't the ES6 `class` syntax **just make it harder** to see and understand the difference between traditional classes and delegated objects?

`class` syntax *does not* provide a way to declare class member properties (only methods). So if you need to do that to track shared state among instances, then you end up going back to the ugly `.prototype` syntax, like this:

```
class C {
    constructor() {
        // make sure to modify the shared state,
        // not set a shadowed property on the
        // instances!
        C.prototype.count++;

        // here, `this.count` works as expected
        // via delegation
        console.log( "Hello: " + this.count );
    }
}

// add a property for shared state directly to
// prototype object
C.prototype.count = 0;

var c1 = new C();
// Hello: 1

var c2 = new C();
// Hello: 2

c1.count === 2; // true
c1.count === c2.count; // true
```

The biggest problem here is that it betrays the `class` syntax by exposing (leakage!) `.prototype` as an implementation detail.

But, we also still have the surprise gotcha that `this.count++` would implicitly create a separate shadowed `.count` property on both `c1` and `c2` objects, rather than updating the shared state. `class` offers us no consolation from that issue, except (presumably) to imply by lack of syntactic support that you shouldn't be doing that *at all*.

Moreover, accidental shadowing is still a hazard:

```
class C {
    constructor(id) {
        // oops, gotcha, we're shadowing `id()` method
        // with a property value on the instance
        this.id = id;
    }
    id() {
        console.log( "Id: " + id );
    }
}

var c1 = new C( "c1" );
c1.id(); // TypeError -- `c1.id` is now the string "c1"
```

There's also some very subtle nuanced issues with how `super` works. You might assume that `super` would be bound in an analogous way to how `this` gets bound (see Chapter 2), which is that `super` would always be bound to one level higher than whatever the current method's position in the `[[Prototype]]` chain is.

However, for performance reasons (`this` binding is already expensive), `super` is not bound dynamically. It's bound sort of "statically", as declaration time. No big deal, right?

Ehh... maybe, maybe not. If you, like most JS devs, start assigning functions around to different objects (which came from `class` definitions), in various different ways, you probably won't be very aware that in all those cases, the `super` mechanism under the covers is having to be re-bound each time.

And depending on what sorts of syntactic approaches you take to these assignments, there may very well be cases where the `super` can't be properly bound (at least, not where you suspect), so you may (at time of writing, TC39 discussion is ongoing on the topic) have to manually bind `super` with `toMethod(..)` (kinda like you have to do `bind(..)` for `this` -- see Chapter 2).

You're used to being able to assign around methods to different objects to *automatically* take advantage of the dynamism of `this` via the *implicit binding* rule (see Chapter 2). But the same will likely not be true with methods that use `super`.

Consider what `super` should do here (against `D` and `E`):

```
class P {
    foo() { console.log( "P.foo" ); }
}

class C extends P {
    foo() {
        super();
    }
}

var c1 = new C();
c1.foo(); // "P.foo"

var D = {
    foo: function() { console.log( "D.foo" ); }
};

var E = {
    foo: C.prototype.foo
};

// Link E to D for delegation
Object.setPrototypeOf( E, D );

E.foo(); // "P.foo"
```

If you were thinking (quite reasonably!) that `super` would be bound dynamically at call-time, you might expect that `super()` would automatically recognize that `E` delegates to `D`, so `E.foo()` using `super()` should call to `D.foo()`.

Not so. For performance pragmatism reasons, `super` is not *late bound* (aka, dynamically bound) like `this` is. Instead it's derived at call-time from `[[HomeObject]].[[Prototype]]`, where `[[HomeObject]]` is statically bound at creation time.

In this particular case, `super()` is still resolving to `P.foo()`, since the method's `[[HomeObject]]` is still `C` and `C.[[Prototype]]` is `P`.

There will *probably* be ways to manually address such gotchas. Using `toMethod(..)` to bind/rebind a method's `[[HomeObject]]` (along with setting the `[[Prototype]]` of that object!) appears to work in this scenario:

```
var D = {
    foo: function() { console.log( "D.foo" ); }
};

// Link E to D for delegation
var E = Object.create( D );

// manually bind `foo`s `[[HomeObject]]` as
// `E`, and `E.[[Prototype]]` is `D`, so thus
// `super()` is `D.foo()`
E.foo = C.prototype.foo.toMethod( E, "foo" );

E.foo(); // "D.foo"
```

Note: `toMethod(..)` clones the method, and takes `homeObject` as its first parameter (which is why we pass `E`), and the second parameter (optionally) sets a `name` for the new method (which keep at "foo").

It remains to be seen if there are other corner case gotchas that devs will run into beyond this scenario. Regardless, you will have to be diligent and stay aware of which places the engine automatically figures out `super` for you, and which places you have to manually take care of it. **Ugh!**

Static > Dynamic?

But the biggest problem of all about ES6 `class` is that all these various gotchas mean `class` sorta opts you into a syntax which seems to imply (like traditional classes) that once you declare a `class`, it's a static definition of a (future instantiated) thing. You completely lose sight of the fact `C` is an object, a concrete thing, which you can directly interact with.

In traditional class-oriented languages, you never adjust the definition of a class later, so the class design pattern doesn't suggest such capabilities. But **one of the most powerful parts** of JS is that it *is* dynamic, and the definition of any object is (unless you make it immutable) a fluid and mutable *thing*.

`class` seems to imply you shouldn't do such things, by forcing you into the uglier `.prototype` syntax to do so, or forcing you think about `super` gotchas, etc. It also offers *very little* support for any of the pitfalls that this dynamism can bring.

In other words, it's as if `class` is telling you: "dynamic is too hard, so it's probably not a good idea. Here's a static-looking syntax, so code your stuff statically."

What a sad commentary on JavaScript: **dynamic is too hard, let's pretend to be (but not actually be!) static.**

These are the reasons why ES6 `class` is masquerading as a nice solution to syntactic headaches, but it's actually muddying the waters further and making things worse for JS and for clear and concise understanding.

Note: If you use the `.bind(..)` utility to make a hard-bound function (see Chapter 2), the function created is not subclassable with ES6 `extend` like normal functions are.

Review (TL;DR)

`class` does a very good job of pretending to fix the problems with the class/inheritance design pattern in JS. But it actually does the opposite: **it hides many of the problems, and introduces other subtle but dangerous ones**.

`class` contributes to the ongoing confusion of "class" in JavaScript which has plagued the language for nearly two decades. In some respects, it asks more questions than it answers, and it feels in totality like a very unnatural fit on top of the elegant simplicity of the `[[Prototype]]` mechanism.

Bottom line: if ES6 `class` makes it harder to robustly leverage `[[Prototype]]`, and hides the most important nature of the JS object mechanism -- **the live delegation links between objects** -- shouldn't we see `class` as creating more troubles than it solves, and just relegate it to an anti-pattern?

I can't really answer that question for you. But I hope this book has fully explored the issue at a deeper level than you've ever gone before, and has given you the information you need *to answer it yourself*.

You Don't Know JS: *this* & Object Prototypes

Appendix B: Acknowledgments

I have many people to thank for making this book title and the overall series happen.

First, I must thank my wife Christen Simpson, and my two kids Ethan and Emily, for putting up with Dad always pecking away at the computer. Even when not writing books, my obsession with JavaScript glues my eyes to the screen far more than it should. That time I borrow from my family is the reason these books can so deeply and completely explain JavaScript to you, the reader. I owe my family everything.

I'd like to thank my editors at O'Reilly, namely Simon St.Laurent and Brian MacDonald, as well as the rest of the editorial and marketing staff. They are fantastic to work with, and have been especially accommodating during this experiment into "open source" book writing, editing, and production.

Thank you to the many folks who have participated in making this book series better by providing editorial suggestions and corrections, including Shelley Powers, Tim Ferro, Evan Borden, Forrest L. Norvell, Jennifer Davis, Jesse Harlin, and many others. A big thank you to Nick Berardi for writing the Foreword for this title.

Thank you to the countless folks in the community, including members of the TC39 committee, who have shared so much knowledge with the rest of us, and especially tolerated my incessant questions and explorations with patience and detail. John-David Dalton, Juriy "kangax" Zaytsev, Mathias Bynens, Axel Rauschmayer, Nicholas Zakas, Angus Croll, Reginald Braithwaite, Dave Herman, Brendan Eich, Allen Wirfs-Brock, Bradley Meck, Domenic Denicola, David Walsh, Tim Disney, Peter van der Zee, Andrea Giammarchi, Kit Cambridge, Eric Elliott, and so many others, I can't even scratch the surface.

The *You Don't Know JS* book series was born on Kickstarter, so I also wish to thank all my (nearly) 500 generous backers, without whom this book series could not have happened:

> Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh,

> simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsden, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel בר-לבב Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuț Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney,

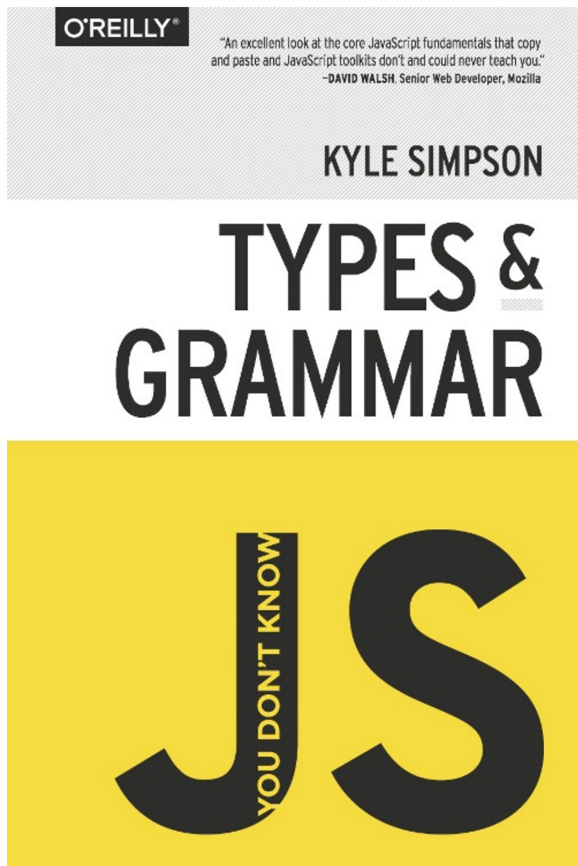
> Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave ♥♫★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry,

> Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard

This book series is being produced in an open source fashion, including editing and production. We owe GitHub a debt of gratitude for making that sort of thing possible for the community!

Thank you again to all the countless folks I didn't name but who I nonetheless owe thanks. May this book series be "owned" by all of us and serve to contribute to increasing awareness and understanding of the JavaScript language, to the benefit of all current and future community contributors.

You Don't Know JS: Types & Grammar



Purchase digital/print copy from O'Reilly

Table of Contents

You Don't Know JS: Types & Grammar

Foreword

It was once said, "JavaScript is the only language developers don't learn to use before using it."

I laugh each time I hear that quote because it was true for me and I suspect it was for many other developers. JavaScript, and maybe even CSS and HTML, were not a core computer science language taught at college in the Internet's early days, so personal development was very much based on the budding developer's search and "view source" abilities to piece together these basic web languages.

I still remember my first high school website project. The task was to create any type of web store, and me being a James Bond fan, I decided to create a Goldeneye store. It had everything: the Goldeneye midi theme song playing in the background, a JavaScript-powered crosshairs following the mouse around the screen, and a gunshot sound that played upon every click. Q would have been proud of this masterpiece of a website.

I tell that story because I did back then what many developers are doing today: I copied and pasted chunks of JavaScript code into my project without having a clue what's actually happening. The widespread use of JavaScript toolkits like jQuery have, in their own small way, perpetuated this pattern of nonlearning of core JavaScript.

I'm not disparaging JavaScript toolkit use; after all, I'm a member of the MooTools JavaScript team! But the reason JavaScript toolkits are as powerful as they are is because their developers know the fundamentals, and their "gotchas," and apply them magnificently. As useful as these toolkits are, it's still incredibly important to know the basics of the language, and with books like Kyle Simpson's *You Don't Know JS* series, there's no excuse not to learn them.

Types and Grammar, the third installment of the series, is an excellent look at the core JavaScript fundamentals that copy and paste and JavaScript toolkits don't and could never teach you. Coercion and its pitfalls, natives as constructors, and the whole gamut of JavaScript basics are thoroughly explained with focused code examples. Like the other books in this series, Kyle cuts straight to the point: no fluff and word-smithing -- exactly the type of tech book I love.

Enjoy Types and Grammar and don't let it get too far away from your desk!

David Walsh
http://davidwalsh.name, @davidwalshblog
Senior Web Developer, Mozilla